SPECTRUM Machine Code

SHIVA's friendly micro series

Ian Stewart and Robin Jones

Spectrum Machine Code

Spectrum Machine Code

Ian Stewart
Mathematics Institute, University of Warwick

Robin Jones
Computer Unit, South Kent College of Technology

Shiva Publishing Limited

SHIVA PUBLISHING LIMITED
4 Church Lane, Nantwich, Cheshire CW5 5RQ, England

Reprinted 1984

ISBN 0 906812 35 6

Cover photograph of the ZX Spectrum kindly supplied by Sinclair Research Ltd.

Typeset and printed by Devon Print Group, Exeter

Contents

Preface

This is an introduction to Z80 Machine Code designed specifically for the ZX Spectrum. It assumes that you know a reasonable amount of BASIC, but absolutely nothing about Machine Code; and by way of simple examples and projects it takes you to the point where you can write your own Machine Code routines, run them within a BASIC program, SAVE them on tape, and LOAD them back in. It is essentially the Spectrum version of the second half of our book *Machine Code and better Basic*, revised to take the particular features of the Spectrum into account, and with additions that bring more of the Spectrum's special abilities into play. As we said in that book, the principles of good programming and of Z80 Machine Code are the same for *any* computer that uses a Z80 CPU. That's true; but of course it does make life easier to work from a description that is absolutely tailored to your own machine. So, to dot the i's and cross the t's, and save you the bother of adapting listings, it's all done for you here.

The main advantage of Machine Code is that it can carry out a number of tasks which are *possible* in BASIC but run too slowly to be acceptable. The main disadvantage is that it imposes many more demands on the programmer, who is required to keep track of the minute bookkeeping details of exactly whereabouts in the machine the information is stored, what form it takes, and how the Spectrum will interpret it. But, as a bonus, if you *do* learn Machine Code then you will also learn a lot about your machine!

We begin by setting up some "theory": how numbers are stored in the computer, how negative numbers are handled, and how binary and hexadecimal codes (which are indispensible) work. Next, we study the architecture of a *simplified* version of the Z80 chip, to establish the main principles (registers, addressing modes, indexing and indirection, the program counter and stack pointer) without having to worry about finicky details. If you start right away on the Z80, every statement made has to be qualified with ifs and buts and maybes: it's a sophisticated animal, and it's a lot easier to see how it works by comparison with something simpler.

Next, the actual Z80 is described, and one important group of commands—the LOAD instructions—is discussed in detail. This group is used to explain the different "addressing modes" in Machine Code.

We explain how to store, run, save, and load Machine Code, and develop a BASIC program to make all of these tasks as easy as possible. All subsequent programs in the book make use of this BASIC program.

The routines discussed include a Machine Code multiplier (which exemplifies many important commands). We describe in detail how the Spectrum's display is controlled, and how it can be manipulated; separate chapters examine useful Machine Code routines for the Attributes File and Display File (including things like scrolling, colour-changing, pattern-generation, switching FLASH on and off).

The most useful flags are described in more detail, using as an example a line-renumbering routine. Two powerful groups of commands, block search and block transfer, are introduced, and the latter applied to various scrolling routines. A final chapter fills in some noteworthy extras.

Appendices include several tables of useful information for Machine Code programming: hex-to-decimal conversion; memory reservation; system variables in hex; the

Z8Ø commands; their effects on Carry and Zero flags; hex codes (in a convenient alphabetical listing); and a useful partial assembler (HELPA) written in BASIC for easy modification, which lets you write, edit, and run Machine Code in a reasonably painless way. In particular, it calculates relative jumps automatically, saving a lot of fiddly calculations (which tend to go wrong, causing havoc, if you do them by hand).

The book is written in such a way that you can, if you wish, make good use of the Machine Code routines *without* understanding how they work. But we hope that you'll aim for something more satisfying: learning how to write your *own* Machine Code.

So far we've referred to ourselves as "we"; but as in our other books, we found this wouldn't work out very well later. So from now on, we'll refer to ourselves as "I". Whenever we (I) say "we", it will mean "I and the reader". It may sound a silly idea, but it's actually more informative that way.

Just to prove that Machine Code really will do things that aren't possible in BASIC, here is a program to produce spectacular, if rather pointless, displays.

1 To Whet Your Appetite...

You wouldn't have bought this book, or be thumbing through it in a bookshop, unless you'd heard that the Spectrum can do remarkable things, quickly, in something called *Machine Code*. Now that's true; but the trouble with Machine Code is that, unlike BASIC, it doesn't do your thinking for you. You have to pay much more attention to finicky details, and keep an eye on exactly whereabouts in the machine your code sits. Machine Code is emphatically *not* "User-Friendly" and to begin with looks rather like Egyptian hieroglyphs, and has the appeal and immediate comprehensibility of an Urdu telephone directory.

It's not really quite that bad, and you'll soon pick it up; but you'll certainly need to put in quite an effort before you come to the real payoff. So, to convince you that it will all be worthwhile, I'm going to start with some Machine Code routines that produce dramatic high-speed moving displays, of a rather abstract character. If you can get BASIC to do this, then sorting out the World Economy is obviously the kind of thing you do on your head before breakfast (which no doubt consists of *three* Shredded Wheat . . .).

Don't try to understand how all this works: that comes later. Just copy, and RUN. You'll need a few keys that you probably haven't made much use of, notably

USR (key "L" in extended mode)
CLEAR (key "X" in keyword mode)

and

POKE (key "O" in keyword mode).

Here's the first program.

```
1Ø  CLEAR 31999
2Ø  BORDER Ø
3Ø  DATA 1, Ø, 3, 17, Ø, 88, 33, Ø, Ø, 237, 176, 2Ø1
4Ø  FOR i = Ø TO 11
5Ø  READ x
6Ø  POKE 32ØØØ + i, x
7Ø  NEXT i
8Ø  PAUSE Ø
9Ø  LET y = USR 32ØØØ
```

Be careful to get this right, especially the DATA list. Now RUN it. The screen will stay blank until you press a key (because of line 8Ø). Press: you get an *instant* response, and the screen fills with coloured squares, some flashing. (If not, recheck the listing. You may have to pull the plug first to reset: that's one of the snags of Machine Code.)

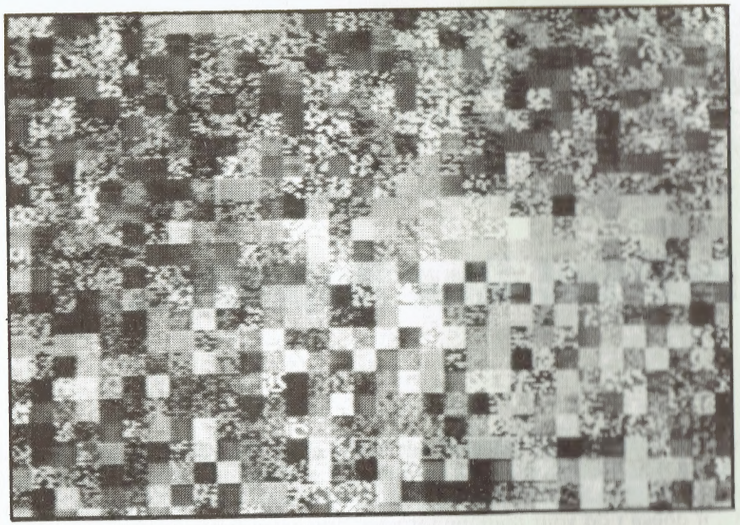


Figure 1.1 Sidescrolling squares in random colours, some speckled, some bright, some flashing . . .

That's fast, but not very dramatic yet. The next step is to make a few changes in the program. Delete line 90 and add:

```
100  LET t = 0: LET s = 0
110  IF t > = 256 THEN LET t = t - 256 * INT (t/256):
     LET s = s + 1
120  POKE 32007, t: POKE 32008, s
130  LET y = USR 32000
140  LET t = t + 1
150  GO TO 110
```

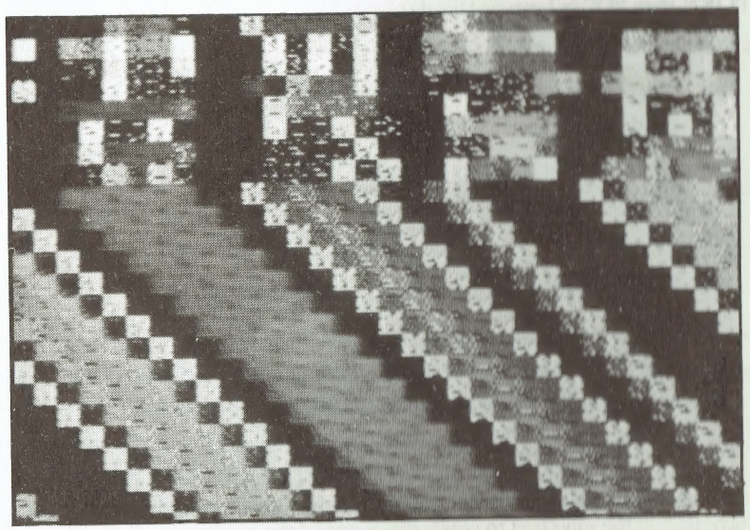


Figure 1.2 . . . acquiring more structure now: rapidly rotating patterns of stripes . . .

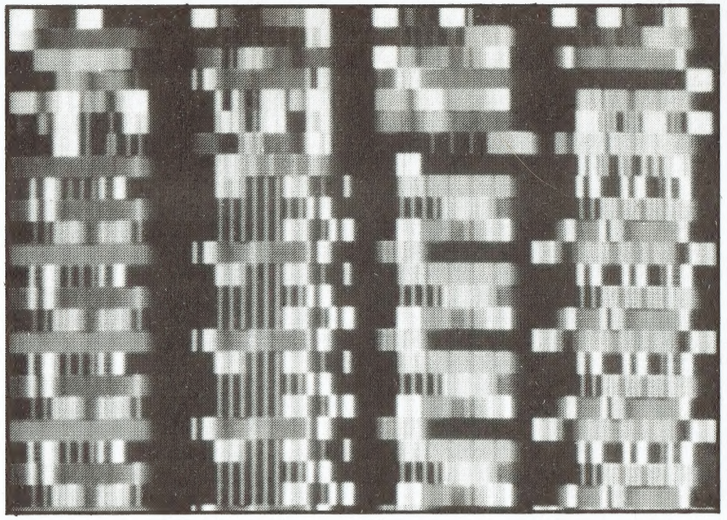


Figure 1.3 . . . bet you won't spot this one! They go by so fast . . .

RUN, and hit a key to start it off. You get a similar display, but now moving *sideways* at a reasonable scrolling sort of speed. Change line 140 to

```
140  LET t = t + 32
```

and it scrolls upwards; change to

```
140  LET t = t + 31
```

and it scrolls *diagonally*. Try a few other numbers to add to t.

Next, change 140 back to LET t = t + 1 and alter the DATA statement to

```
30  DATA 1, 0, 24, 17, 0, 64, 33, 0, 0, 237, 176, 201
```

and repeat the procedure. I won't tell you what to expect: see for yourself! Make changes to line 140 as before.

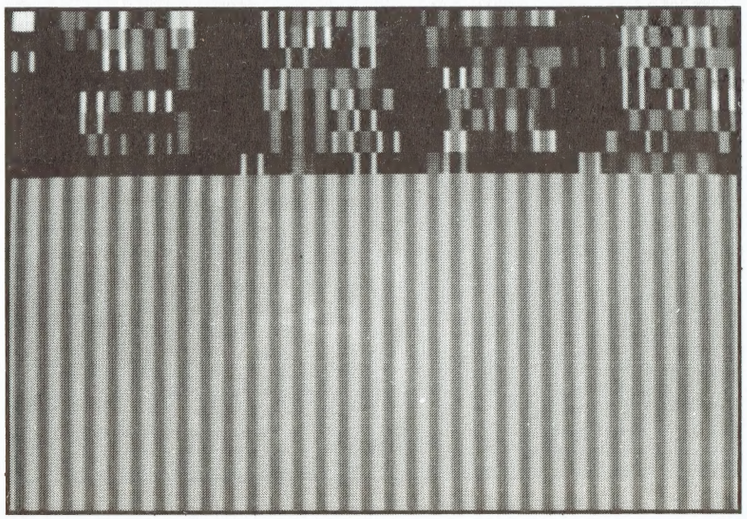


Figure 1.4 . . . now a pause, with gentle green bands . . .

LIGHT-SHOW

Unusual, maybe; perhaps not spectacular yet. Now we combine both routines together, with a little fine-tuning:

```
10  CLEAR 31999
20  BORDER 0
30  DATA 1,0,24,17,0,64,33,0,0,237,176,201
35  DATA 1,0,3,17,0,88,33,0,0,237,176,201
40  FOR i = 0 TO 23
50  READ x
60  POKE 32000 + i, x
70  NEXT i
100 LET t = 0: LET s = 60
110 IF t >= 256 THEN LET t = t - 256 * INT (t/256):
    LET s = s + 1
120 POKE 32007, t: POKE 32008, s
125 POKE 32019, t: POKE 32020, s - 1
130 LET y = USR 32000
140 LET y = USR 32012
150 LET t = t + 1
160 GO TO 110
```

RUN this, and keep it running for a minute or two: it starts peacefully enough, but then all Hell breaks loose . . .

Change line 150 to t + 32, t + 31, etc. as before.

Change the initializations in line 100. What happens if you start with s = 0? 40?

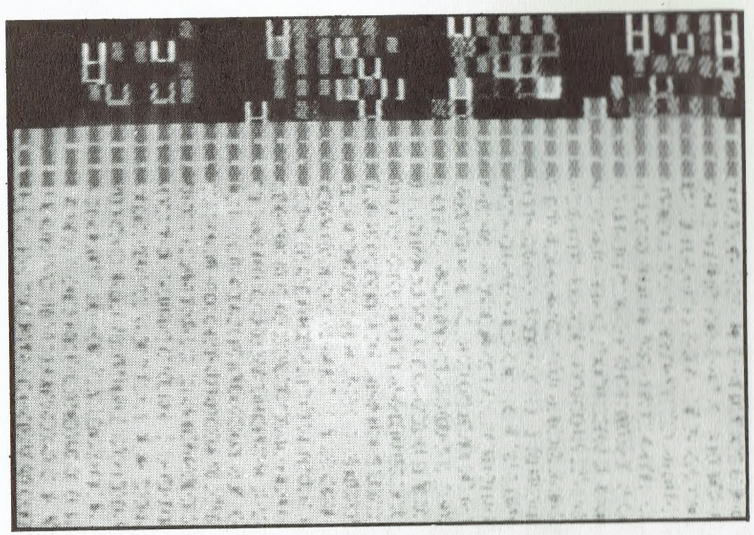


Figure 1.5 . . . here they come, like hordes of angry insects, eating everything in their path!

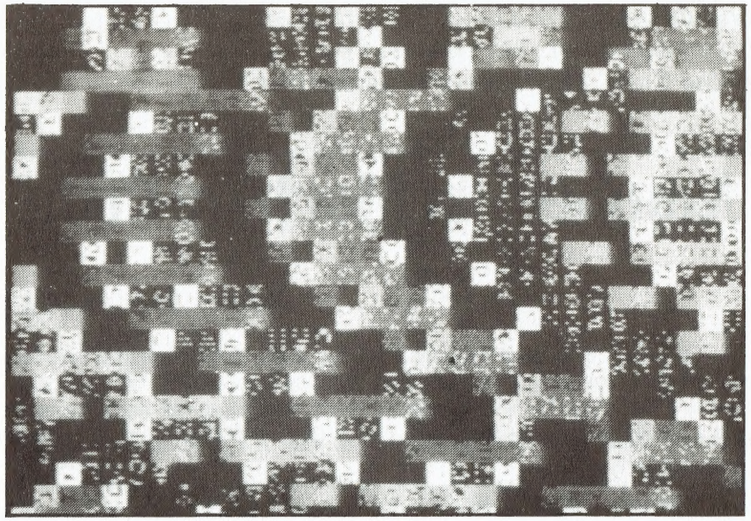


Figure 1.6 . . . and on into massive swirls of colour. All from two dozen bytes of Machine Code (and 16K of ROM). And these are only samples!

The action gets so fast around s = 63 or so that it's hard to follow. If you add the line

```
145 IF INKEY$ = " " THEN PAUSE 0
```

you'll find that nothing happens if you aren't pressing a key, and you can "single-step" the program at any point by depressing the key quickly and taking your finger away, then repeating this. You'll see a lot of patterns that passed by too quickly during the previous mad scramble.

You can invent endless variations on this program: just don't touch lines 30, 35, 130 or 140, where the Machine Code stuff is dealt with.

I hope you're convinced by now that Machine Code does have something extra to offer. Though, apart from producing pretty patterns at high speed, it would be hard to maintain that this particular example does anything especially *useful*. Its main advantage is that it is a very short piece of Machine Code with a disproportionate effect. To make good use of Machine Code routines, we will need to be able to write them in a properly structured way, and to aim at a specific purpose (just like a good BASIC program). The remainder of this book is devoted to that aim.

The binary system is a bit like counting on your feet instead of your fingers. For Machine Code, all you need is sixteen feet.

2 Numbers in Machine Code

We normally think about numbers in terms of tens. If I write the number 3814 we all understand that to mean:

$$3 \times 1000 + 8 \times 100 + 1 \times 10 + 4 \times 1$$

and we can see that to get a "place value" from the one on its right we simply multiply by ten. We say the number is in *base* ten.

Because we've been doing this for as long as we can remember, it's difficult to realize that there are other, perfectly sensible, ways of doing the same job. Early computer designers certainly didn't; they used base ten representations in their machines and hit some nasty snags. Mostly, they were caused by the fact that electronic amplifiers don't behave the same way for all the signals you want to input to them. For instance, an amplifier that is supposed to output double its input signal may well do so for inputs of 1, 2, 3 and 4 units; but then it starts to "flatten off" so that an input of 5 produces an output of only 9.6, 6 produces 10.8, and maybe you can hardly tell the difference between the outputs for inputs of 8 and 9.

Put a music tape in your cheapo cassette recorder and wind up the volume. Hear the distortion in the loud bits? It's the same effect.

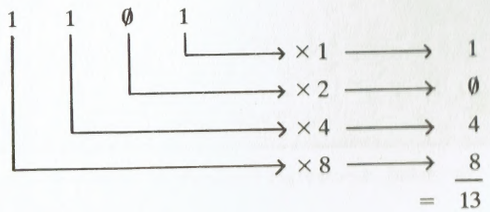
Pioneer computer designers didn't hear any distortion; they just found that the machines couldn't distinguish between different digits at times, and that was hopeless for a computer. So they had to rethink their number representation to suit what the electronic gubbins would do best.

The simplest thing you can do with an electrical signal is to turn it on or off; so you can represent the digits 0 (off) and 1 (on) satisfactorily. Distortion no longer matters. It's clear whether a signal is present or not regardless of how mangled it is. But can we devise a number system which only uses 0s and 1s?

Yes. In a base ten number, the largest possible digit is 9. Add 1 to 9 and you get 10—a *carry* has taken place. We can write any number using any other base we choose, and the largest possible digit will always be one less than the base. If the base is 2, the largest digit is 1, so a base 2 (or *binary*) number only contains 0s and 1s.

What about the place values? In the base ten case we got those by starting at 1 (on the right) and multiplying by 10 every time we moved left one place. For a binary number we still start at 1, but we multiply by 2 every time we move left.

So for instance the binary number 1101 can be converted to base 10 like this:



Converting the other way is easy as well; take 25 for example. If we write down the binary place values:

32 16 8 4 2 1

and work from the left, it's clear that we need a 16, which leaves 9, and that's made up of an 8 and a 1, so 25 is:

0 1 1 0 0 1

HEXADECIMAL CODE

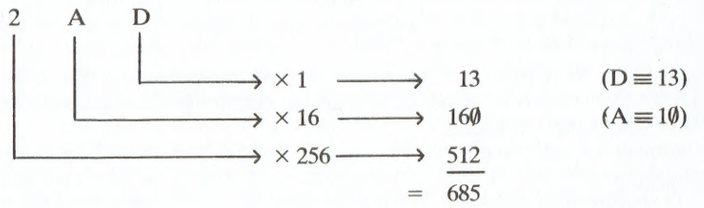
This is fine for relatively small values, but a bit messy for large ones. There are a number of quick conversion techniques, and there are binary-to-decimal and decimal-to-binary conversion program listings in *Easy Programming for the ZX Spectrum*; but I want to examine a procedure which makes use of *hexadecimal* code, because it will stand us in good stead later.

A number in hex (nobody ever says "hexadecimal", except me, just now) is a number in base 16. So the place values are obtained by successive multiplications by 16. The first five are:

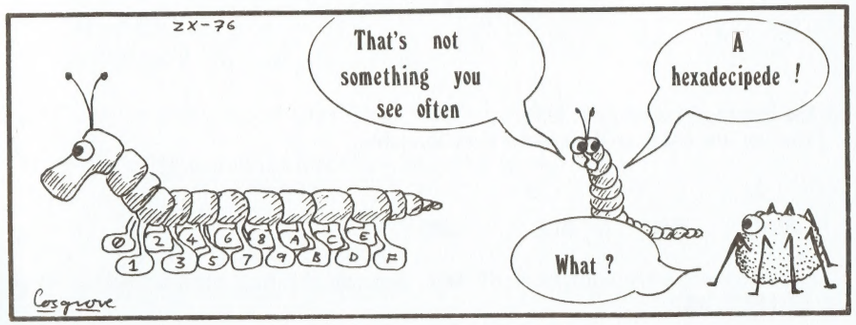
65536 4096 256 16 1

"Hang about!" everybody's saying. "Those are nasty numbers, and anyway, in base 16 the largest digit has the value 15. Things are getting complicated."

Bear with me. We handle the problem of digits greater than 9 by assigning the letters A–F to the values 10–15. So the number 2AD in hex converts to decimal like this:



Now for the nice feature of hex. Because 16 is one of the binary place values (the fifth one) it turns out that each hex digit in a number can be replaced by the four binary digits which represent it. (By the way, "binary digit" takes almost as long to say as "hexadecimal" so it's normally abbreviated to "*bit*".) The table overpage shows the conversions.



Decimal	Hex	Binary
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
10	A	1010
11	B	1011
12	C	1100
13	D	1101
14	E	1110
15	F	1111

A more extensive table is given in Appendix 1. (*Note*: In this book, as it is written for the Spectrum, the letters may be upper or lower case; but inputs will assume *lower* case (no CAPS LOCK) for convenience.)

Now suppose we want to convert 9041 to hex. First we extract two 4096s, then some 256s and so on like this:

```
          9041
2 × 4096 = 8192 −
            849
3 ×  256 =  768 −
             81
5 ×   16 =   80 −
              1
1 ×    1 =    1 −
              0
```

So the hex representation is 2351.

Now we just copy the digit codes from the table:

```
  2     3     5     1
0010  0011  0101  0001
```

and that's the binary equivalent of 9041; just run the four blocks together to get 0010001101010001.

The hex-to-binary conversion is so easy that, more often than not, we leave numbers in hex even when, ultimately, we need them in binary. After all, it's easy to make an error in copying long strings of 0s and 1s.

CONVERSION BY COMPUTER

Here's a program to convert from decimal to hex. It successively divides the number by 16, looking at the remainder each time; so it extracts digits in the opposite order to that shown above.

```
20 LET p = 4
30 LET h$ = "0000"
40 INPUT "dec. no. (max 65535)"; dn
50 LET n = INT (dn/16)
60 LET r = dn − 16 * n
70 LET h$ (p) = CHR$ (r + 48 + 39 * (r > 9) )
80 LET dn = n
90 LET p = p − 1
100 IF dn > 0 THEN GO TO 50
110 PRINT "Hex value is:"; h$
```

Line 70 arranges for the digits and letters a–f (we choose to use *lower case*) to be selected. It looks a bit confusing because the ASCII code used by the Spectrum to represent characters internally behaves in an inconvenient way. We want to count . . . 7, 8, 9, a, b, . . . and it would be nice if the code for "a" were 1 greater than that for "9". Unfortunately, it's actually 40 greater, that is, 39 too big. So we have to add the 39 in for characters greater than 9. The logical expression "r > 9" has value 1 if true, 0 if false; so the extra 39 gets added in when the character is in the a–f range. (The 48 is needed because the ASCII code for 0 is 48.)

The result is always presented as a 4-digit number with leading zeros as required, and the letters in lower case. This is because we'll always *enter* hex digits in lower case, so it serves as a reminder. The program won't work if the result should contain more than four hex digits, but that's fine for our purposes because of the limits to memory size in the Spectrum—only four hex digits are required anyway.

Here's the code to convert the other way (hex to decimal).

```
140 INPUT "Enter 4-digit number in hex"; h$
150 LET dn = 0
160 FOR p = 1 TO 4
170 LET dn = dn * 16 + CODE h$ (p) − 48 − 39 * (h$ (p) > "9")
180 NEXT p
190 PRINT "Decimal value is:"; dn
```

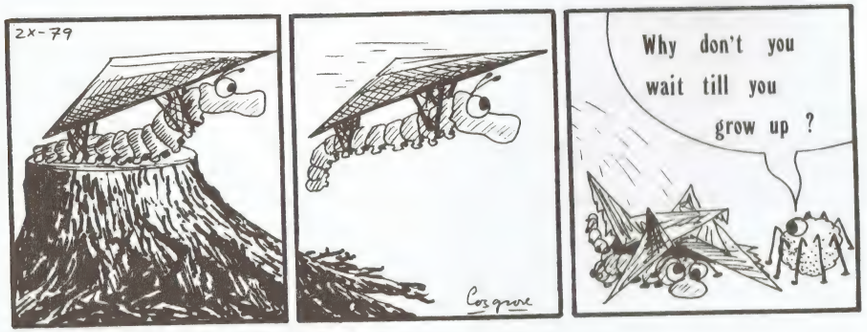
Again there's a fiddle to generate the right values on line 170 which is just the reverse of line 70.

We could tie these routines together with a little menu:

```
2 PRINT "Dec/Hex converter"
3 PRINT "1) DEC − > HEX
         2) HEX − > DEC
```

```
        3) END"
  5  INPUT "Enter 1, 2, or 3"; sel
  8  IF sel = 1 THEN GO SUB 2Ø
  9  IF sel = 2 THEN GO SUB 14Ø
 1Ø  IF sel = 3 THEN STOP
 12  PAUSE Ø: GO TO 2
```

Of course we'll need RETURNs on lines 12Ø and 2ØØ.



To deal with negative numbers, the machine uses a clever trick.

3 Positive and Negative

Now that we've seen something about manipulating binary numbers let's return to looking at the way they are handled inside the machine. Usually, a number is held in a fixed number of bits, often 16 or 24 or 32, depending on the machine design. This number of bits is called the *word size* for the machine.

Let's examine what numbers could be held in a 4-bit word:

4-bit pattern	Decimal value
ØØØØ	Ø
ØØØ1	1
ØØ1Ø	2
ØØ11	3
Ø1ØØ	4
Ø1Ø1	5
Ø11Ø	6
Ø111	7
1ØØØ	8
1ØØ1	9
1Ø1Ø	1Ø
1Ø11	11
11ØØ	12
11Ø1	13
111Ø	14
1111	15

It's obvious why bigger word sizes are chosen in practice; a machine which can only represent the numbers Ø to 15 is unlikely to be adequate. But there are two other problems; the notation can't represent fractional values (7.14, for instance) and it can't represent negative numbers.

We'll ignore the fractions problem because most Machine Code routines only use integers, but the way in which negative numbers are dealt with is more pressing.

The technique is simple: if you've got the binary representation of a positive number and you want to create its negative equivalent you do two things:

1. Change all the Øs to 1s and all the 1s to Øs (this is rather picturesquely called "flipping the bits").
2. Add 1 to the result.

For instance, suppose you want −3.

3 = ØØ11 in a 4-bit word

Flipping the bits gives: 11ØØ
Now add 1: +1
11Ø1

So 11Ø1 represents −3. It's called the *2's complement* of ØØ11.

I'm not going to explain exactly why this works, but you can prove to yourself that it does in any particular case like this:

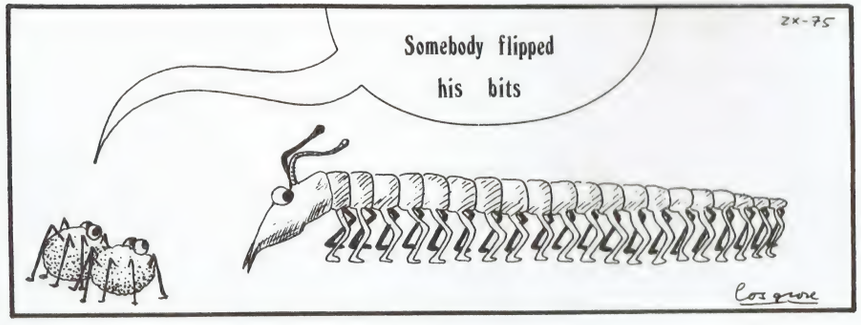
If we add 3 to −3 (or 5 to −5 or anything to minus itself) we should get zero. So:

```
   ØØ11     ( = 3)
+  11Ø1     ( = −3)
=  1ØØØØ
   ----
   111      (Don't forget that 1 + 1 = Ø carry 1 in binary!)
```

So we *don't* get ØØØØ at all; but the junior 4 bits *are* zero, and if we're working in a 4-bit word the senior bit will just drop off the end. (For a convenient analogy, think about a car trip-meter with 3 digits; if it reads 999 and you drive an extra mile, it reads ØØØ and a "1" has "dropped off" the left hand end).

In other words we should have seen it like this:

```
  [ØØ11]
+ [11ØØ]
  ----
  [ØØØØ]
    ↓
    1
```



This always works provided that the number of bits is fixed throughout. Don't forget to include leading zeros to make up the number of bits to this standard length, *before* taking the 2's complement.

Let's rewrite the 4-bit table of values, now including negatives:

Decimal	Binary	2's complement	Decimal
Ø	ØØØØ	ØØØØ	Ø
1	ØØØ1	1111	−1
2	ØØ1Ø	111Ø	−2
3	ØØ11	11Ø1	−3
4	Ø1ØØ	11ØØ	−4
5	Ø1Ø1	1Ø11	−5
6	Ø11Ø	1Ø1Ø	−6
7	Ø111	1ØØ1	−7
8	1ØØØ	1ØØØ	−8
9	1ØØ1	Ø111	−9
1Ø	1Ø1Ø	Ø11Ø	−1Ø
11	1Ø11	Ø1Ø1	−11
12	11ØØ	Ø1ØØ	−12
13	11Ø1	ØØ11	−13
14	111Ø	ØØ1Ø	−14
15	1111	ØØØ1	−15

Straight away we see that there's a problem; every bit-pattern occurs twice so that, for instance, 1ØØ1 could mean 9 or −7. So we'll have to restrict the range of values still further. I've drawn a dotted line around the region we actually choose to represent. If you look at the senior (leftmost) bit in each of the patterns you'll notice that it's "Ø" if the number is positive and "1" if the number is negative. This is obviously a very convenient distinction.

So the range of numbers we can get into a 4-bit word is −8 to +7. For 5 bits it would be −16 to +15. For 6 bits it will be −32 to +31 and so on.

A 16 bit word (which is important so far as the Z80 is concerned) holds the range −32768 to +32767. A table of 2's complement notations for 8-bit words is given in Appendix 1.

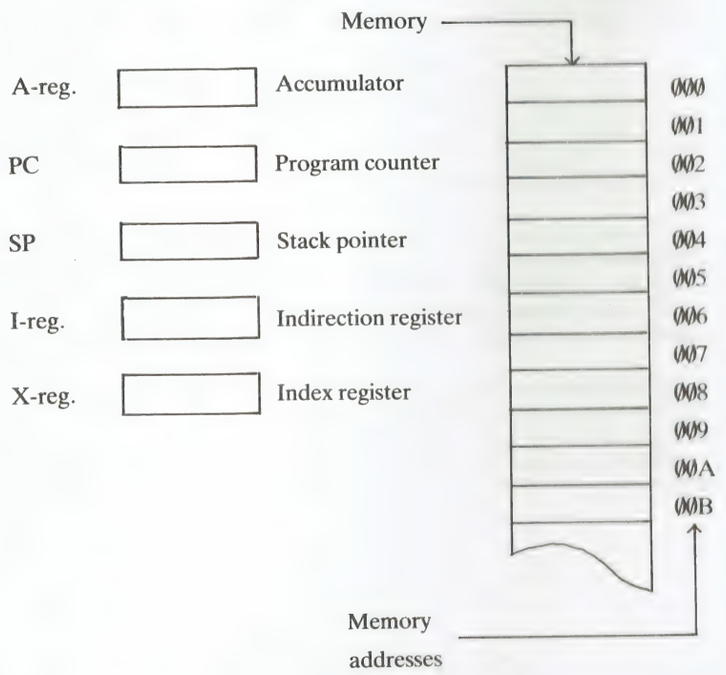
It's easier to start with a simplified, imaginary machine. The Z80 is like this, but more complicated: get the main ideas here!

4 Machine Architecture

That's enough about numbers. Now we'll look at how the machine crunches them. To do this, we need to know about the internal structure of the processor—its *architecture*.

Now, the Z80 processor is the product of some twenty-five years of computer development and is a fairly sophisticated beast. So it's not really a good place for the beginner to start. What I'm going to do, then, is describe a simple processor which might have been built in the late 1940s (except it wasn't), just to introduce the important concepts which are relevant to virtually all current devices, without having to worry about the frills, which we can look at later (in Chapters 7 onwards).

We'll suppose that our imaginary machine has a memory of 16-bit words and a number of 16-bit special-purpose registers as shown below:



Let's look at the memory first. In BASIC we could have called each of those memory locations anything we fancied, but the naked machine isn't so friendly. It insists on numbering every location in an absolutely fixed way, starting at zero, as I've shown.

These numbers are called the *memory addresses*, and I've numbered them in hex, although you should always bear in mind that, ultimately, the coding will be binary.

What can be held in a memory word? Well, any pattern of 16 bits. Obvious; but the point I'm driving at is that those 16 bits can mean anything we want them to mean. If we want them to mean a 2's complement coded integer then a word holds a number in the range −32768 to 32767. If we want them to mean a positive integer with no sign bit then the number is in the range Ø to 65535. If we want, we can split the word into two 8-bit fields each of which represents an alphabetic, punctuation or graphics symbol. As Tweedledee (or was it Tweedledum?)* said: "When *I* use a word, it means just what I choose it to mean—neither more nor less." I sometimes think Lewis Carroll was ahead of his time.

Now for the special-purpose registers. Just the A-register to kick off with. This is used every time you do any arithmetic. The result of any sum you ask the machine to do is put into the A-register. (Sometimes it's called the *accumulator*, by the way.) Most arithmetic operations work on two values; it's no good asking the machine to work out 3+, you need to say what 3 is to be added to. One of these values must be in the A-register before the addition operation is executed. So you can write an instruction like:

ADD (1A3)

and the machine takes that to mean:

1. Add the contents of memory location 1A3 to the contents of the A-register. (The brackets round 1A3 are being used to indicate that it's the *contents* of 1A3 and not the *number* Ø1A3 which is to be added.)
2. Put the result back in the A-register.

We've just written our first machine level instruction. It's not actually in Machine Code, but it's close. Look at its general form. It consists of an operation code, ADD, and an address, (1A3). Many instructions will look like that. Incidentally, life is too short to say "operation code" too often; everybody shortens it to *opcode*.

AN ADDITION PROGRAM

Let's think about a sequence of machine instructions which would model the BASIC statement:

LET R = B + C

First we would have to assign actual addresses to R, B and C. Suppose that these are 1Ø3, 1Ø4 and 1Ø5, respectively. We have to get the contents of 1Ø4 into the A-register. Let's invent an LD (for load accumulator) instruction to do this:

LD (1Ø4)

then add on the contents of 1Ø5

ADD (1Ø5)

and finally we need a way of storing the A-register's contents back in 1Ø3. So we'll invent a "store" instruction:

ST (1Ø3)

Now we have a simple machine level program consisting of 3 instructions:

LD (1Ø4)	[load B into A-register]
ADD (1Ø5)	[add on C]
ST (1Ø3)	[put the result in R]

How do we get the machine to run such a program?

* See page 79.

We're used to the idea that a program is stored in the machine *before* it's executed. After all, if you wrote the BASIC statement:

```
10  PRINT "HELLO WORLD"
```

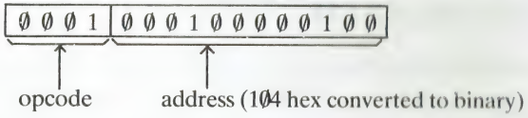
you'd be somewhat disconcerted if, as soon as you hit ENTER, the message "HELLO WORLD" were displayed. You expect it to be held until you need it. So, by the same token, a machine level program has to be stored first. Where more natural to store an instruction than in a memory word? (A word means what you want it to mean—remember?) Of course, that implies that the opcodes LD, ADD and so on have to be coded as bit patterns, but all we have to do is invent a table of bit patterns in a quite arbitrary way like this:

Opcode mnemonic	Binary code
ADD	0000
LD	0001
ST	0010

and every time we think of a new opcode that's needed, we add it to the table.

I've assumed, above, that all opcodes have a 4-bit binary code. That allows 16 different patterns and therefore 16 distinct instructions. This is a small instruction set by modern standards but it will do for our hypothetical toy computer. We've got 16 bits in the word altogether, so 12 are left for the address portion of the instruction.

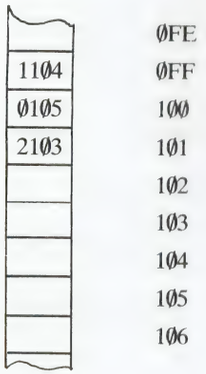
So LD (104), once inside the machine looks like:



Once you've seen one bit pattern, you've seen them all, so from now on we'll write the hex versions of instructions. It's marginally less tedious.

THE PROGRAM COUNTER

Suppose we store our 3-instruction program from location 0FF onwards:



Now we need a way of saying to the machine: "Kick things off by executing the instruction in 0FF, then do the one in 100, then one in 101." That's what the PC-register, or *program counter*, is for. It acts as a kind of bookmark for the computer. We run the program by initializing the PC to the address of the first instruction. While the machine is obeying this instruction, the PC is automatically updated by 1, so that when the system returns to examine the PC, it will go and obey the next instruction, and so on.

There's a snag, though. While the last instruction (in 101) is being dealt with, the PC will be updated by 1 as usual, and so when the machine looks at it again, it will find 102, and leap off to execute the instruction there. What instruction? We didn't put one in 102. Ah! But there has to be a bit-pattern in 102 left by a previous program, or just set up when the machine was switched on. So the machine will interpret this pattern as if it is an instruction, because that's what we've asked it to do. And then it will roll on through locations 103, 104 and 105 and that's where we're storing data! So if the number in 104 is 20FF, for instance, the machine will interpret this as:

```
ST (0FF)
```

which will copy the contents of the A-register into 0FF, thereby destroying the first instruction of our program! Obviously what we need is a "halt" instruction (I'll use the mnemonic HLT) which stops the updating of the PC in its tracks. So the program now reads:

```
LD (104)
ADD (105)
ST (103)
HLT
```

There's an important point here. Precisely *because* we are using words to mean different things at different times, we have to keep a very careful eye on the implications the machine will draw from what we tell it to do. If we request it to ADD the contents of a location to the A-register, then it will assume that that location holds a number. It will make no tests; it cannot—any bit-pattern could represent a number. Similarly, any bit-pattern could represent an instruction, so if the PC points to a location, its contents will be executed as an instruction.

The rule is: *keep data and programs firmly apart*. If you don't, you can expect to be totally mystified at regular intervals. As I've indicated, a whole program can disappear without trace while it is running!

Some more instructions: the functions of the program counter and the stack.

5 Jumps and Subroutines

So far, our instruction set looks a bit thin. We've got LD and ST, which will move things around memory, ADD, which is pretty primitive arithmetic, and we can stop things with HLT.

We'll pep up the arithmetic capability a bit by adding SUB, which will subtract the contents of a location from the A-register, but that's all we're getting. No multiply, no divide, definitely no square root.

What we really need is a set of branch instructions, equivalent to BASIC's IF ... THEN ...

JUMPS

It's going to be fairly easy to branch to an instruction out of the usual sequence; what we need to do is change the contents of the PC. So we'll use an instruction like:

```
JP 416          [jump to 416]
```

Whenever it is executed, it will put 416 in the PC. The system is "fooled" into thinking that the next instruction is in 416, and then it will go on to 417, 418 etc. until the next "jump" instruction is encountered. Of course, any address can follow the JP opcode.

This instruction is more like a GO TO than an IF ... THEN .... What we need is an instruction which resets the PC only if some condition is met. The simplest test we can make is whether the A-register contains zero.

```
JPZ 2A7         [jump to 2A7 only if A-reg. contains Ø]
```

Another would be:

```
JPN 14E         [jump to 14E only if contents of A-reg. are negative]
```

That's the minimum we can get away with, because we can now test for a positive (non-zero) number by noticing when the program doesn't jump on *either* JPZ or JPN instructions.

SUBROUTINES AND STACKS

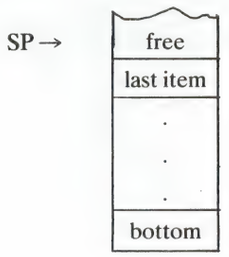
While we're on the subject of transferring control from one place to another inside the program, how about something like BASIC's GO SUB and RETURN?

We'll have an instruction:

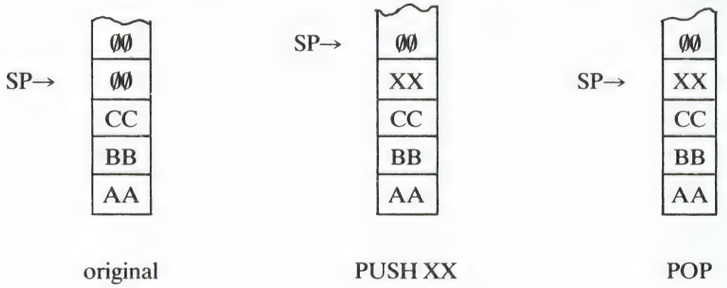
```
CALL 2Ø5        [call the subroutine starting in 2Ø5]
```

What does it do? Well, obviously it puts 2Ø5 into the PC, but we could use a JP for that.

CALL performs a second function: it stores the address of the instruction after the CALL, so that when a "return" (opcode: RET) is encountered it can load the stored address back into the PC to continue the main program from where it left off. This is done by using a *stack*. A stack is a segment of memory with a fixed "bottom" and a variable top. A stack pointer SP holds the address of the top: it is called a "pointer" because it points to the top of the stack, like this:



Extra items can be PUSHed on to the stack by moving SP up one and putting the new item in memory; and they can be POPped off the stack by reducing the SP by one. (It's not actually necessary to delete the POPped item from memory: the stack routines ignore anything above the SP.) For example:



In fact the SP points to the first *unused* location. (Further, the Z8Ø machine stack in the Spectrum points downwards, not up; but don't worry about that, as it makes no difference to the user, who is never required to know the actual stack addresses.

Stacks work on the principle "Last in, first out". Imagine a pile of books on a desk. PUSH means "add a book to the top of the pile" and POP (effectively) means "take a book off the top". So if you PUSH three items X, Y, Z in turn

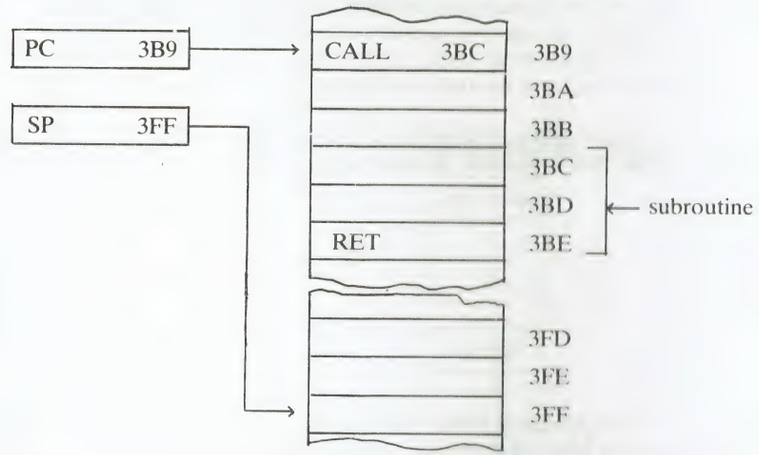
```
PUSH X
PUSH Y
PUSH Z
```

then to get them off in the right order you need to use

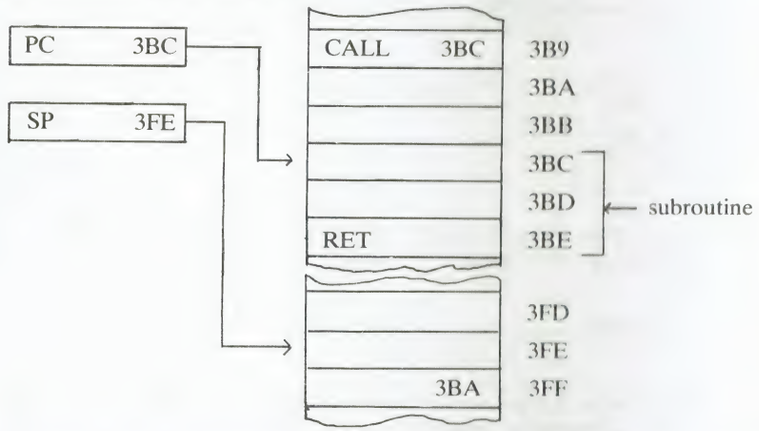
```
POP Z
POP Y
POP X
```

A subroutine uses the stack to remember its return address. When a CALL is obeyed, the return address (the address of the CALL + 1) is pushed on to the stack. When the

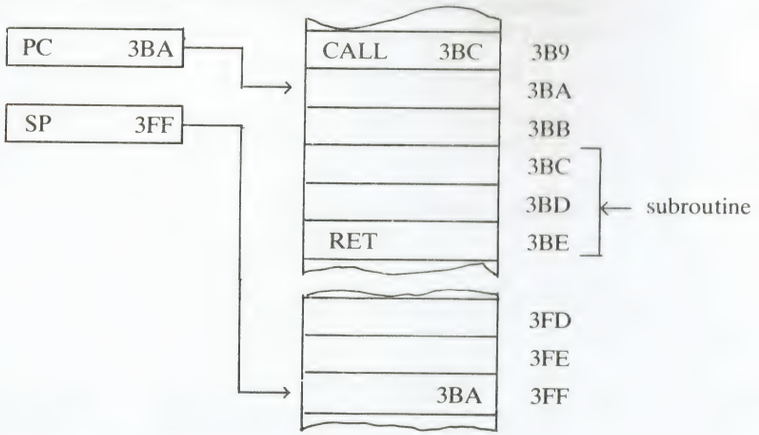
RET is encountered the stack is popped into the PC. Here's an example:



The CALL is about to be obeyed . . .



Now it has been, and the return address is on the stack. The program steps through the subroutine until it reaches the RET, after which:



and control is back inside the main program.

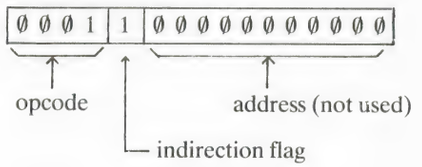
The private eye tracks his victim: how to use the contents of one address to point to another one.

6 Indirection and Indexing

There are only two registers left to talk about, and both have similar functions: they can both alter the address part of an instruction while the program is running.

INDIRECTION

Let's look at the way the I-register does this first. We'll invent a new opcode, LDI or "load indirect". Like HLT, it doesn't have an address associated with it. To the machine, it's just like an LD except that the high bit of the address field is set to "1". This bit is called the *indirection flag*, and simply indicates to the machine that indirection is in force. So the binary form of the LDI instruction is:



The hex code is 18ØØ. When the machine encounters this instruction, it uses whatever number is in the I-register as the effective address. So if the I-register contains 1E4 and an LDI instruction is executed, the effect is exactly the same as if the instruction had been LD 1E4. In other words the I-register acts as a memory pointer, and we can move it around to our heart's content if we can do arithmetic with it. That means moving values into the A-register, because that's the only place we can do arithmetic. So we'll invent an opcode XAI for "exchange contents of A-register with contents of I-register".

Of course, the indirection flag can be set for any instruction which has an address part. So we can have STI, JPI, ADDI etc. and in each case, the last three digits of the hex code will be 8ØØ.

AN EXAMPLE

Let's look at an example which uses these ideas. Suppose that we want to initialize a 1D array of length 2Ø, to hold the numbers 2, 4, 6, 8 . . . 4Ø. In other words we want a machine code equivalent of the BASIC:

```
FOR c = 1 TO 2Ø
LET a (c) = c * 2
NEXT c
```

There are a series of values which are going to have to be in memory somewhere, to make this work. They are 1 (because the loop count goes up in ones), 2 (because that's the increment for the array contents) and 20 (which is needed to test for the end of the loop). I don't, for the moment, want to be bothered with exactly where these numbers should be stored, so I'm going to allow addresses to be referred to temporarily by names (just like BASIC names). We'll have to convert these to numbers when we finally get to machine code, of course. This is an application of Jones's First Law of Computing: "Never put off till tomorrow what you can put off till the day after." So we'll assume that the numbers we want are available in locations called N1, N2 and N20. Similarly, we'll have a location called BASE which holds the address of the first element of the array, and one called COUNT which will act as the loop counter.

First we set the I-register to point to the base of the array:

```
LD   BASE
XAI
```

Then we set the COUNT to 1:

```
LD   N1
ST   COUNT
```

Now we double this (by adding it back into the A-register) and store it in the location pointed at by the I-register. (We talk about "storing *through* the I-register" for short.)

```
ADD  COUNT
STI
```

We "undouble" the value on the A-register again, subtract 20 and see if the result is zero. If it is we've finished:

```
SUB  COUNT
SUB  N20
JPZ  OUT
```

OUT is another, as yet unspecified, address. We don't know where it is yet, because we don't know where the program ends, and so, again, it's useful to give it a name temporarily.

If the branch doesn't occur, we add 1 to the COUNT:

```
LD     COUNT
ADD    N1
ST     COUNT
```

and increment the I-register by 1:

```
XAI
ADD    N1
XAI
```

The current COUNT is now back in the A-register, so we can loop back to the doubling operation:

```
JP     LOOP
```

provided we give the "ADD COUNT" instruction the symbolic address "LOOP". Let's do this by preceding the instruction by its symbolic address followed by a colon:


```
LOOP:  ADD    COUNT
```

We can do the same sort of thing to set up the initial values we need, by defining a new opcode HEX which just sets a word to a required value. It isn't really an opcode at all since it isn't equivalent to a machine instruction, so we call it a pseudo-operation. The whole program looks like this (ignore the numbers in the left- and right-hand margins for the moment):

```
020        LD      BASE      1  033
021        XAI                A  000
022        LD      N1        1  030
023        ST      COUNT     2  032
024 LOOP:  ADD     COUNT     0  032
025        STI               2  800
026        SUB     COUNT     4  032
027        SUB     N20       4  031
028        JPZ     OUT       6  047
029        LD      COUNT     1  032
02A        ADD     N1        0  030
02B        ST      COUNT     2  032
02C        XAI                A  000
02D        ADD     N1        0  030
02E        XAI                A  000
02F        JP      LOOP      5  024
030 N1:    HEX     0001      0  001
031 N20:   HEX     0014      0  014
032 COUNT: HEX     0000      0  000
033 BASE:  HEX     0033      0  033
```

The only symbolic address which doesn't appear in the left-hand column, and is therefore still unspecified, is OUT. We'll worry about it later.

The form of the program we now have is written in what is known as *assembly code*. On modern sophisticated computers there will be an *assembler program* whose function is to convert this into real machine code for us.

HAND ASSEMBLY

Alas, neither our hypothetical machine nor the Spectrum has such a program (although commercial ones can be bought). So we have to do the job by hand. We need a table of opcodes and their equivalent hex values:

Opcode	Hex
ADD	Ø
LD	1
ST	2
HLT	3
SUB	4
JP	5
JPZ	6
JPN	7
CALL	8
RET	9
XAI	A

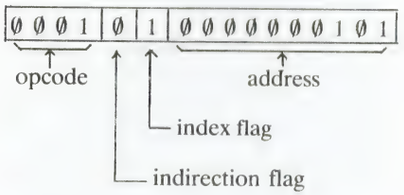
Also we need to know where the beginning of the program is. That's a more or less arbitrary decision, so let's assume it's at Ø2Ø. Since each instruction occupies 1 word, we can write down the address of each instruction. You'll see that I've done this down the left-hand side of the program on page 25. Now we can replace the opcodes and addresses by their hex equivalents. For instance, LD BASE becomes 1 Ø33, since BASE is now identified as Ø33. The right-hand margin on page 25 shows the complete code.

The only instruction which needs further comment is JPZ OUT, which encodes as 6 Ø47. Why should OUT be at Ø47? It could be elsewhere, but Ø47 is the first location it *can* be at. The reason is that the array is occupying the space from Ø33 to Ø46 (twenty words), and we obviously don't want to go clumping around inside the program's data area.

THE INDEX REGISTER

When the X-register is in use, the real instruction address is formed by adding the address field to the contents of the X-register. For instance, if the X-register contains 4ØØ, then the instruction LDX ØØ5 has the same effect as LD 4Ø5.

We'll pinch another bit of the address field to indicate when indexing is in operation, so the LDX ØØ5 instruction looks like this:



In hex, that's 14Ø5. Actually there's nothing you can do with indexing that you can't do with indirection. It's just that it will do arithmetic with addresses automatically instead of leaving the job to you.

The actual architecture of the Z80 CPU, the heart (or is it the brain?) of your Spectrum.

7 At last the Z80!

I'm sorry that you've had to wade through the last ten or so pages without being able to try anything out, but if you've really understood the ideas in them, you'll find that understanding the Z80 is a breeze.

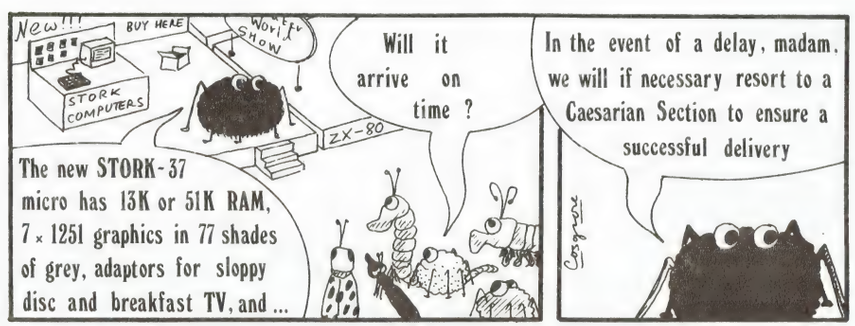
Before we get into the Z80's architecture (sorry, the chapter heading isn't quite accurate!) let's consider some of the difficulties of the processor I've just described.

First, the 4-bit operation code only allows 16 different instructions. (OK, we cheated a little, by allowing the indirection and indexing flags to spill over into the address field, but that in turn means we've limited the address size, and therefore the maximum size of memory!) The Z80 has 694 instructions! To give each of them a separate bit pattern means that we need an 8-bit field (1 byte); and even then some fudging is needed.

Second, our imaginary machine uses memory in a rather careless way. Some of the instructions don't use the address field (HLT, LDI, STI, for instance), so a sequence of such instructions wastes 1Ø bits in every word. The Z80 gets over this problem by allowing different instructions to have different lengths. Some instructions have no address field and are just 1 byte long. Others have a 1-byte address field and so are 2 bytes long. Others have a 2-byte address field for a total of 3 bytes. There are even some which have 2-byte opcodes! This means that the PC can't increment by 1 for every instruction executed. It has to increment by the length of the instruction.

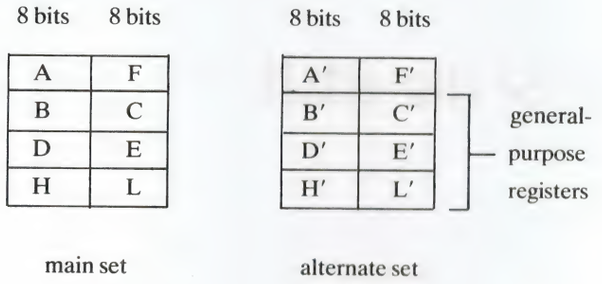
Third, we always have to handle 16-bit words, which is inconvenient if we're dealing with characters (which normally occupy a byte each). So it would be nice to allow 8-bit *and* 16-bit operations.

Fourth, the fact that there is only one general-purpose register (the A-register) can be annoying. It often means that intermediate results have to be stored temporarily back in memory while some other calculation is done. The Z80 has a number of general-purpose registers; although, as we shall see, exactly how many there are varies depending on what we're using them for.



THE REGISTERS

Here's the register organization:



Ignore the alternate set for the moment.

The registers appear in pairs, indicating that they may be used either as 8-bit or 16-bit registers. For instance, we can refer to the B-register (8 bits), or the C-register (8 bits) or the BC register (16 bits). The B, C, D, E, H and L registers can all be used in this way (in pairs BC, DE, HL *only*) but the A and F registers are strictly 8-bit registers and cannot be combined. For the 16-bit pairs, the senior byte is the left-hand one (B, D, H) as you'd expect.

There are two index registers, the IX and IY, a stack pointer (SP) and program counter (PC). What, no indirection? Actually any of the 16-bit general-purpose register pairs (BC, DE or HL) can be used for indirection but, for simplicity, we shall always use the HL for this purpose.

There are two sets of instructions, one for handling 8-bit operations and the other for handling 16-bit operations. We'll start with the 8-bit "load" instructions.

There are five different ways to refer to the address used in Z80 commands. They are most easily understood with reference to the LD group of commands, which shuttle data from one place to another.

8 Addressing Modes and the LD Instructions

Let's look at the "load" (LD) operation as an example of the 8-bit group. It's very like the LD instruction in our imaginary machine, except that two extra addressing modes are allowed: *register-to-register*, and *immediate*. That gives a total of five, with *direct*, *indirect* and *indexed* available as before.

1. *Direct addressing*
   This looks much the same as our imaginary equivalent, except that, since there is more than one register, we have to specify which register we want loaded:

   LD A, (ØF1C)

   This loads the contents of ØF1C into the A-register. Note that, by convention, the movement is from right to left, so that we can write:

   LD (ØF1C), A

   and mean "copy the contents of the A-register into ØF1C".

   (Actually, the A-register is the only *8-bit* register which can be directly addressed.)

2. *Indirect addressing*
   Again, this is straightforward. Since we're going to standardize on the HL for indirection, the instruction format is:

   LD A, (HL)

   which means "load the A-register *through* (i.e. from the address contained in) the HL register". To pass data in the opposite direction we could have:

   LD (HL), A

   which puts the contents of A into the address *contained in* HL. (For this instruction, registers other than A are allowed.)

3. *Indexed addressing*
   Here, we need to indicate which index register is in use, and the amount of the offset:

   LD A, (IX + 2E)

   Note that in direct addressing, I showed an address of 4 hex digits, because 16 bits (2 bytes) are allowed for the address. The offset value in an indexed address instruction must be held in 1 byte, however, so I've only shown two hex digits.

4. *Register-to register*
   We can transfer data between registers like this:

   LD D, B

   which means: "load the contents of B into D".

5. *Immediate*
   Here, data itself, rather than the address of data, is placed in the address field. So we can write:

   LD B, Ø7

   to mean "put the number 7 in B". Note again that the number is two hex digits, since it has to be stored in the single byte of the B-register. Note also that a "LD" is really a *copy*: the numbers are retained in their original addresses or registers, but a copy is placed at the destination.

HEX CODES

Now let's see what each of these instructions looks like in hex; for a full listing see Appendix 6:

1. LD A, (ØF1C)

First we look up the opcode for the LD A, (nn) instruction. (The nn indicates a general 2 byte address). This is 3A. So you would expect the instruction to code as:

3A ØF 1C

Unfortunately, there's a slight complication caused by the way the Z80 thinks about numbers; it likes the least significant (junior) byte of an address first. So we have to *swap the address bytes round*:

3A 1C ØF

This is mildly annoying, but you soon get used to it. It is an invariable rule for 2-byte numbers in Z80 instructions: *junior byte first, then senior*: hence all those PEEK X + 256 * PEEK (X + 1)'s in the Sinclair *Manual*.

The LD (nn), A instruction has the code 32, so:

LD (ØF1C) becomes 32 1C ØF

2. LD A, (HL)

This is easy. There is no address part so it's just a 1-byte opcode. Look it up and you'll find it's 7E.

Similarly LD (HL), A codes as 77.

3. LD A, (IX + 2E)

The general instruction is LD A, (IX + d), d indicating a 1-byte displacement (in 2's complement notation), and its code is DD 7E. (Note that—a 2-byte opcode!) So the instruction is:

DD 7E 2E

where the byte 2E is the displacement chosen in this case.

4. LD D, B

No problem here, again. The code is 5Ø.

5. LD B, Ø7

The opcode is Ø6 so the instruction is Ø6 Ø7.

As well as learning to write *Machine Code, we're going to have to learn how to get the Spectrum to carry it out, and how to record our efforts for posterity.*

9 Storing, Running, and Saving Machine Code

The main aim of this chapter will be to develop a simple BASIC utility program to take some of the pain out of Machine Code. A more sophisticated version of this is given in Appendix 7.

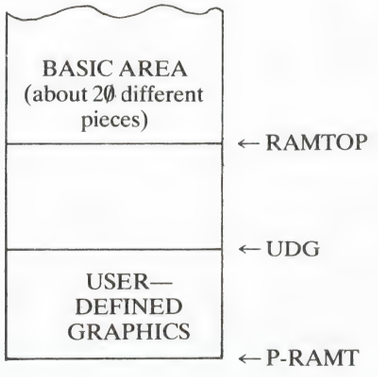
Let's start with the Spectrum's memory. It comes in two kinds:

Type	Start address		Finish address		
	Decimal	Hex	Decimal	Hex	
ROM	Ø	Ø	16383	3FFF	
RAM	16384	4ØØØ	32767	7FFF	(16K machine)
			65535	FFFF	(48K machine)

The ROM holds the operating system, BASIC interpreter, and so on: it would be disastrous if we could tamper with it, and it's designed so that we can't: ROM means "Read Only Memory". It can be PEEKed but not POKEd.

The RAM (Random Access Memory) is ours to do with as we wish; and it is in RAM that the Machine Code must be stored and executed. One potential problem is that the BASIC system also wants to use the RAM: if we store the Machine Code in any old position the BASIC system may wipe it out, overwrite it, or generally clobber it. We need to keep the Machine Code somewhere not used (or at least not altered) by the BASIC system.

The civilized place (others are possible, see later) is at the top end of memory. Here's a picture of the relevant memory area.



The addresses of the boundaries between these regions are held in three system variables, and the addresses of *those* are the same in 16K or 48K:

RAMTOP	23730–1
UDG	23675–6
P-RAMT	23732–3

These are two-byte variables (see *Further Programming*,* page 33) and you can find the value of (e.g.) RAMTOP by using

```
PRINT PEEK 23730 + 256 * PEEK 23731
```

When the machine is first switched on, these are set to standard values:

Variable	16K		48K	
	Decimal	Hex	Decimal	Hex
RAMTOP	32599	7F57	65367	FF57
UDG	32600	7F58	65368	FF58
P-RAMT	32767	7FFF	65535	FFFF

The BASIC system uses the area below *and including* the address in RAMTOP. The part from UDG up to P-RAMT (top of physical RAM) stores the format for user-defined graphics. This area can be reduced in size or cleared out altogether by changing the value of UDG; but to avoid complicating the issue I'll assume that we want to leave the user-defined graphics area in its standard place.

We make room for the Machine Code by lowering the value of RAMTOP. For example, to release 600 bytes of space, and start the Machine Code at address 32600, we must change RAMTOP to 32600 − 1 = 31999.

(This gives more than enough space for everything in this book, and the start address for Machine Code is a round number both in decimal (32600) and hex (7D00), so I'll standardize on that value. If you want a different value, just change every occurrence of 32600 and 31999 or 7D00 and 7CFF to whatever you prefer. To avoid having to deal separately with the 48K machine I'll *also* use 32600 for that; but I'm wasting about 32K of RAM by doing so. You may prefer to use 64000, which is FA00 in hex. While *learning* Machine Code, the wasted space is irrelevant; but in actual programs you'd certainly want to choose RAMTOP more efficiently.)

To lower RAMTOP, use the following command:

```
CLEAR 31999
```

This has the following effects:

It clears all variables.
It clears the display file, like a CLS would.
It resets the PLOT position to lower left: (0, 0).
It RESTOREs the DATA pointer to the start.
It clears out the GO SUB stack, holding the return addresses of the subroutines so far used.
It resets RAMTOP to 31999, making addresses 32600 and upwards "safe" from interference.
It places the new GO SUB stack underneath this new value of RAMTOP.

A command like

```
CLEAR 31447
```

would do exactly the same things, but reset RAMTOP to 31447, leaving a lot more room: addresses 31448 and upwards would be "safe".

* *Further Programming for the ZX Spectrum* by Ian Stewart and Robin Jones, Shiva.

Obviously the best time to use CLEAR is before you've assigned any variables, displayed anything, or called any subroutines. The loading utility developed below resets RAMTOP right at the start, but if you're doing your own thing it's not a bad idea to get into the habit of using CLEAR *before* you do anything else.

Warning

To add emphasis to the above discussion: to make the area from the address "address" upwards safe, use CLEAR address −1, *not* just CLEAR address. The value in a CLEAR command is the *last unsafe* address, not the first safe one. It's a nuisance, but a minor one: make sure you remember it.

STORING MACHINE CODE

What we need is a program that accepts hex codes, and stores the corresponding bytes in order above RAMTOP. The following routine achieves this: it will be added to later in this chapter to make it more versatile.

```
10 CLEAR 31999
20 PRINT "Base Address: 32600"
30 PRINT "Number of Data bytes: □";
40 INPUT d: PRINT d
50 LET b = 32600
60 FOR i = 0 TO d
70 POKE b + i, 0
80 NEXT i
90 LET a = b + d
100 DIM h$ (2)
110 PRINT "CODE:"
120 INPUT c$
130 IF c$ = "s" THEN GO TO 200
140 PRINT c$ + "□";
150 LET hs = CODE c$ (1) − 48 − 39 * (c$ (1) > "9")
160 LET hj = CODE c$ (2) − 48 − 39 * (c$ (2) > "9")
170 POKE a, 16 * hs + hj
180 LET a = a + 1
190 GO TO 120
200 POKE a, 201
```

The aim of line 30 is to let us reserve some addresses in front of the Machine Code program, to hold data in: this is often useful. Lines 60–80 fill this spare space with zeros; the actual data can be POKEd in when needed later. Lines 150–160 convert the hex code into two (decimal) numbers between 0 and 15 (instead of 0–9, A–F) such that 16 * hs + hj gives the actual decimal version of the hex code. Line 170 POKEs this into the correct address.

The input "s", which isn't a hex code, acts as a delimiter, telling the machine to stop asking for the code. Line 200 makes sure that the final command in the Machine Code program is the RET (return to BASIC) instruction, which has hex code C9 (decimal 201). As the *Manual* points out on page 180, it is important to end the Machine Code

routine with this instruction; so the loader above puts it in automatically, just in case you forget. (An extra RET does no harm, so it won't matter if you remember and end up putting it in twice.)

RUNNING MACHINE CODE

To run the routine, the command USR is used. For technical reasons USR is a *function*, whose argument is the address at which the Machine Code routine starts, and whose value is the contents of the Z80's BC-register when the routine ends. Along the way to working this out, however, the Spectrum will actually carry out the Machine Code program. (This may sound a bit perverse, but it's intended to make Machine Code easily accessible within a BASIC program.) If the Machine Code starts at address 32000, we need a command such as

```
LET y = USR 32000
```

although we're seldom interested in the actual value of y at the end of it all! Any command that involves USR 32000 and has correct syntax will do: for example

```
RANDOMIZE USR 32000
```

(note that this is efficient since RANDOMIZE needs only a single keystroke) or

```
RESTORE USR 32000
```

or whatever . . .

If there are data bytes, then the 32000 must be increased to avoid them being treated as part of the program. The following, added to what we already have, lets us run Machine Code. It is set up to allow other options at this stage—see below.

```
300  INPUT "Option?"; o$
310  IF o$ = "r" THEN GO SUB 400
399  STOP
400  LET y = USR (b + d)
410  RETURN
```

SAVING MACHINE CODE

The way to SAVE a Machine Code program is to use *byte storage*, see the *Manual*, page 144. Suppose our routine turns out to be 77 bytes long, including the final RET command. Then we SAVE it, under the name "m-code" (say) with the command

```
SAVE "m-code" CODE 32000, 77
```

where the 32000 is the starting byte and the 77 is the length. In fact, since we've standardized the positioning, it's pointless to worry about the length, and we could replace the 77 by 600 as a standard procedure. (It does take a little more space on the tape, but very little.)

To LOAD back such a set of stored bytes is equally simple: the command

```
LOAD "m-code" CODE
```

will do it. If you've forgotten the name, use

```
LOAD " " CODE
```

We may as well build both of these as options into our loading utility:

```
320  IF o$ = "s" THEN GO SUB 500
330  IF o$ = "l" THEN GO SUB 600
```



```
500  INPUT "Name for SAVEing?"; n$
510  IF n$ = " " THEN LET n$ = "m-code"
520  SAVE n$ CODE b, 600
530  RETURN

600  INPUT "Name for LOADing?"; n$
610  LOAD n$ CODE
620  RETURN
```

CHECKING AND PRINTING

A final addition lets us check the listing on the screen, or even print it out if we've got a printer. (If there are errors, correct by POKEing to those addresses. HELPA, Appendix 7, does this in a more useful way, but for the moment it will suffice.)

```
340  IF o$ = "z" THEN GO SUB 700
350  IF o$ = "p" THEN GO SUB 700

700  FOR i = b TO a
710  LET j = PEEK i: LET js = INT (j/16):
     LET jj = j - 16 * js
720  LET h$ (1) = CHR$ (js + 48 + 7 * (js > 9) )
730  LET h$ (2) = CHR$ (jj + 48 + 7 * (jj > 9) )
740  IF o$ = "p" THEN PRINT i; "□ □" + h$ + "□ □"; j
750  IF o$ = "z" THEN LPRINT i; "□ □" + h$ + "□ □"; j
760  NEXT i
770  RETURN
```

This produces both a decimal and a hex listing, by address, of the program.

Note that the original input prints out the letters a–f in *lower* case, but the above uses *upper* case. If this disturbs you, it's a good exercise to find a way to prevent it. But it's actually rather convenient to use upper or lower case interchangeably, and I shall do so hereafter.

We can use this option to check the listing both before and after a run, provided we add

```
370  GO TO 300
```

so that we have the options all over again. To STOP you could add

```
360  IF o$ = "a" THEN STOP
```

Now the possible options are:

a	STOP
l	LOAD
p	PRINT (to screen)
r	RUN (Machine Code)
s	SAVE
z	COPY (to printer)

The rest of this book will assume you have a copy of the above program SAVEd on tape, ready to use on the Machine Code listings provided.

To kick off, here are some routines for addition and subtraction. What could be simpler?

10 Arithmetic

The key to arithmetic is the existence of ADD and SUB commands, both of which reference the A-register, and can use any of the addressing modes except direct.

For starters, let's write a program to add the numbers 4 and 7 together. We have to load one into the A-register, load the other into the B-register, add them, and put the result somewhere. If we use the LOADER with 1 data byte we can put the sum into address 32000 (7D00). First translate into opcode mnemonics:

Put 4 in the A-reg:	LD A, 04
Put 7 in the B-reg:	LD B, 07
Add them (with the sum ending up in the A-reg):	ADD A, B
Store the result:	LD (7D00), A

Now look up the opcodes in the Appendix: this gives the hex coding

LD A, 04	3E 04
LD B, 07	06 07
ADD A, B	80
LD (7D00), A	32 00 7D

Notice how the 7D00 comes out in the order 00 7D, as I've just mentioned.

Now to load this in. Get the LOADER program into the Spectrum, and RUN. When asked for data bytes, input 1. Then input in turn the hex codes, in the form

3e 04 06 07 80 32 00 7d

(because LOADER prints out in lower case) and terminate with

s

as delimiter (and to add that crucial RET command, which I forgot!).

As option, choose "r" to run.

Fine, but where's the answer? The easiest way to see it is to choose option "p", a listing. The result is then

32000	0B	11
32001	3E	62
32002	04	4
32003	06	6

32004	07	7
32005	80	128
32006	32	50
32007	00	0
32008	7D	125
32009	C9	201

We see the program, in 32001 onwards, including the final RET, C9. We also see the answer to the sum, 11, sitting in 32000, which is where we wanted it to be.

Before going any further, write Machine Code programs to work out:

18 + 66
13 + 17
23 + 6

by copying the above, but changing the 04 and 07 to the new pairs of numbers; check that the sum does go into 32000 in each case.

BETTER USE OF DATA

That's fine, but we don't want to have to write a new program for each pair of numbers, do we?

We could change the 04 and 07 to general numbers m and n using

POKE 32002, m: POKE 32004, n

But by doing so we're mixing up program and data, because the two numbers really ought to be data. It doesn't really matter here; but in a more complicated program it's a nuisance if the program itself gets changed during a run (this is called a *self-modifying* program). First, because you can't re-use it; second, because there is no record left of what it originally was.

So it would be better practice to set up the two numbers to be added as data; load them into the registers as part of the program; add them, and load back the answer into data.

We therefore reserve three data bytes:

32000	First number
32001	Second number
32002	Total

which in hex are 7D00, 7D01, 7D02. The program, starting at 7D03, ought to look like this:

Load first number into A:	LD A, (7D00)
Load second number into B:	LD B, (7D01)
Add them:	ADD A, B
Put the sum into 32002	LD (7D02), A

That's lovely, except that, when you come to look up the hex codes, you'll find there's no command LD B (nn). Bother!

There are ways round this. One is to use indirection via the HL register, so that instead of LD B (7D01) we write

Load HL with the address:	LD HL, 7D01
Load B through HL:	LD B, (HL)

So now the program takes the form

LD A, (7DØØ)	3A ØØ 7D
LD HL, 7DØ1	21 Ø1 7D
LD B, (HL)	46
ADD A, B	8Ø
LD (7DØ2), A	32 Ø2 7D

Load that in, reserving three data bytes: then STOP the program to set up the data using

POKE 32ØØØ, 44: POKE 32ØØ1, 33

to add (say) 44 and 33. Now GO TO 3ØØ to get back into the LOADER, and enter the option "r" to run. Then enter "p" for a listing. The first three entries go

32ØØØ	2C	44
32ØØ1	21	33
32ØØ2	4D	77

and then the rest of the program follows. The sum, 77, is to be found in address 32ØØ2, as we hoped.

Change the data that are POKEd in: see what the results are.

In particular, try

POKE 32ØØØ, 24Ø: POKE 32ØØ1, 1ØØ

You'll find that the computer is convinced that 24Ø + 1ØØ = 84. If this looks peculiar, it is; but it's your fault, not the computer's! The ADD command forgets carry digits. Think about it in binary:

```
24Ø      1 1 1 1 Ø Ø Ø Ø
1ØØ    + Ø 1 1 Ø Ø 1 Ø Ø
       ---------------
       Ø 1 Ø 1 Ø 1 Ø Ø    = 84
       1 1
       |
       1
```

The sum generates a ninth bit, which can't be held in an 8-bit byte, so it falls off the end and the quoted result is too small by the value of the ninth bit, which is 256. And indeed 256 + 84 = 34Ø, the "correct" answer. No check has been made, no helpful error message printed. When you write Machine Code you're on your own. What you don't test for, you don't find out about.

In fact there are ways to deal with carry digits and with this kind of "overflow": see below, under "flags". But there's no need to worry about that problem yet.

SUBTRACTION

Now see if you can modify the program so that it *subtracts* the second number from the first. All you need do is change the

ADD A, B	8Ø

to

SUB A, B	9Ø

and proceed as before. Explore that, make sure that 44 − 33 = 11; and see what effects you can get from overflows (e.g. try 17 − 99).

There are 694 Z80 instructions: here's a selection of the more fundamental and accessible ones, and what they will do.

11 A Subset of Z80 Instructions

I'm not going to describe every one of the 694 opcodes the Z80 has; that would be tedious and unnecessary. (But see Appendix 4.) We'll look at a subset of 3Ø-odd types of instruction (covering about 23Ø actual commands). Unfortunately, not all of them can use all the addressing modes. Here's a quick reference table showing which instructions can use what; the opcodes are given in Appendix 6.

Opcode / Address Mode	LD	ADD ADC SUB SBC AND OR XOR CP	INC DEC SLA SRA SRL	JR JRC JRNC JRZ JRNZ DJNZ	JP	JPZ JPNZ JPC JPNC JPP JPM	LD	ADD ADC SBC	INC DEC PUSH POP
Register	LD r, s	ADD A, r	INC r					ADD HL, r	INC r
Immediate	LD r, n	ADD A, n			JP nn	JPZ nn	LD r, nn		
Direct	LDA, (nn) LD (nn), A						LD HL, (nn) LD (nn), HL		
Indirect	LD A, (HL) LD (HL), A	ADD A, (HL)	INC (HL)		JP (HL)				
Indexed	LD A, (IY + d) LD (IY + d), A	ADD A, (IY + d)	INC (IY + d)	JR d					

8-bit operations — 16-bit operations

The notation in the table needs some explanation. Some of the opcodes will be unfamiliar, but we'll deal with those later. Otherwise, the conventions are:

1. Each entry in the table shows an example of the format of the instruction type. Any of the other opcodes in that column could be substituted.
2. "r" or "s" denotes any register. Whether this is an 8-bit or a 16-bit register depends on which part of the table the instruction is in. For instance, in the LD r, s instruction, r and s are any 8-bit registers (A, B, C, D, E, H or L), but in ADD HL, r "r" is one of BC, DE, HL, SP.
3. "n" is any 8-bit number. "nn" is any 16-bit number.
4. If a register is explicitly stated, as in LD A, (nn), then this is the only register which may be used for this purpose.

   This is a wild oversimplification. Sometimes, other registers are usable, but the point is that the set of instructions I've shown are always OK and you can worry about extending your vocabulary of instructions when you're handling this lot confidently.
5. "d" is any 8-bit number, but it's always added to some 16-bit value. In other words, it's an indexing displacement.

Now let's look at the new opcodes:

AND

This operation takes the contents of the A-register, and another 8-bit field, and examines these, bit by bit. Only if corresponding bits are both "1" does it put a "1" back in this position in the A-register. Otherwise it inserts a "Ø".

For instance, AND A, Ø7 has the following effect:

A-register before the operation:	Ø Ø 1 1 Ø 1 Ø 1
Ø7:	Ø Ø Ø Ø Ø 1 1 1
A-register after the AND:	Ø Ø Ø Ø Ø 1 Ø 1

See how the junior three bits have been transmitted? So you can use AND to select a portion of a byte.

OR

This works in a similar way to AND, but this time, the resulting bit is a "1" if either of the initial bits is a "1". So OR A, Ø5 gives:

A-register before:	Ø 1 Ø Ø 1 Ø 1 1
Ø5:	Ø Ø Ø Ø Ø 1 Ø 1
A-register after:	Ø 1 Ø Ø 1 1 1 1

Now, certain bits are being forced to "1" regardless of their original value.

XOR

Here the initial bit values must be different for the result to be a "1". XOR A, B3 gives:

A-register before:	Ø 1 Ø 1 1 Ø 1 Ø
B3:	1 Ø 1 1 Ø Ø 1 1
A-register after:	1 1 1 Ø 1 Ø Ø 1

It's particularly useful for flipping a register from Ø to 1 and back again. If the A-register contains Ø to start with, every time the instruction XOR A, Ø1 is executed, the value in the A-register will flip. (Ø to 1, back to Ø, back to 1 and so on.)

CP

This is the "Compare" instruction. The contents of the A-register are compared with those of another 8-bit field. That raises a problem, though: how is the result of the comparison signalled?

This is what the F (or *flags*) register is used for. Each bit of the F-register holds some information about the effect of the last instruction to alter them. (Not all instructions do alter them).

The flags which most interest us are the Carry, Zero, Overflow and Sign flags. CP can alter any of these, but the one of most significance here is the Zero flag, which is set if the two values being compared are equal.

If the A-register contents are *less* than those of the compared byte, the Sign flag is set. This is equivalent to saying "the result is negative". This is all you need to know about the flags at the moment: it's an intricate topic if you delve deeper. See Chapter 16 and Appendix 5 for some additional information.

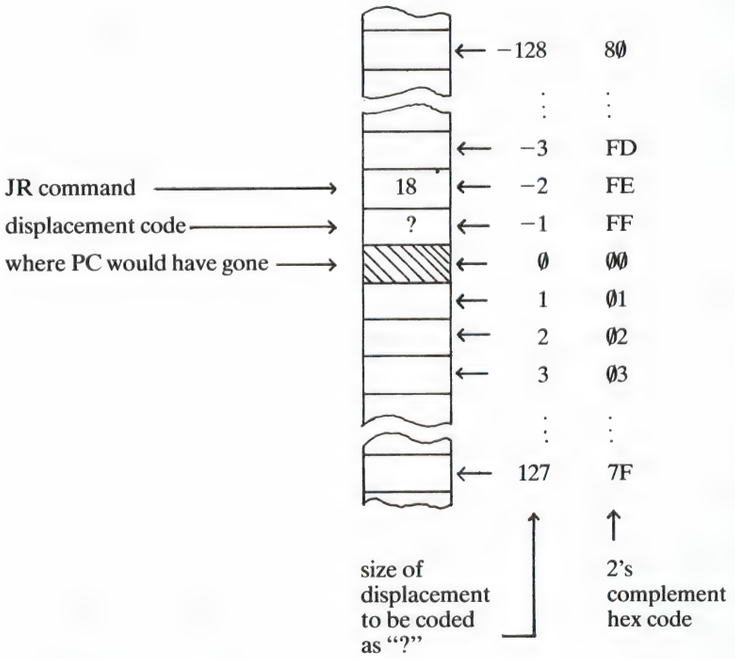
THE JUMPS

All the conditional jumps branch (or not) depending on the contents of the flags. So, for instance, JPZ says "jump if the Zero flag is set". Now we can see how the CP instruction can be used. Suppose, for example, that we wish to see if a particular byte, pointed at by HL, contains 1E hex. If it does, we want to branch to 447B. The code is:

LD A, 1E	3E 1E
CP A, (HL)	BE
JPZ 447B	CA 7B 44

All the other jumps behave similarly; JPNZ says "jump on a non-zero result" (Zero flag *not* set), JPP says "jump on a positive result" (Sign flag *not* set), JPM says "jump on a minus result" (Sign flag set), JPNC says "jump on no carry" (Carry flag *not* set), and so on. All of them have one thing in common, and that is that the address of the jump is fixed. In other words, if, for any reason, we would like a routine to run somewhere in memory other than where we first loaded it, all the jump addresses must be changed. The Z80 deals with this neatly by allowing "relative jumps" (JR). In other words, you can jump so many bytes forward (or back) from where you are. This displacement is held (in 2's complement notation, Appendix 1) in 1 byte, so the distance which can be jumped can't exceed 128 bytes backwards or 127 bytes forwards.

The displacement is calculated from what the PC value *would* have gone to next, had no jump occurred; namely, the address of the next command in the program. Like this:



Here's an example. We want to examine each byte of memory in turn for the first occurrence of 1E hex. Assume for simplicity that the start address is already in HL. We could write:

```
LD A, 1E
```

```
loop:   CP A, (HL)
        INC HL
        JRNZ loop
```

Two points need explaining. First, I've sneaked in a new instruction: INC. This is short for INCrement. It just adds 1 to the contents of the specified register; so the compare operation is always looking at the next memory byte because HL is being bumped up by 1 every loop. (By the way, DEC, short for DECrement, does exactly the opposite.) The second point is that there's no obvious difference between JRNZ *loop* and JPNZ *loop*. It isn't until we *assemble* the instructions into Machine Code that the difference is clear. Suppose the code is to be loaded from 7DØØ hex, as usual:

Address		Instruction	Hex code
7DØØ		LD A, 1E	3E 1E
7DØ2	loop:	CP A, (HL)	BE
7DØ3		INC HL	23
7DØ4		JRNZ loop	2Ø FC

Why is FC in the address part of the JRNZ instruction? It works like this: when the JRNZ instruction is executed the PC is bumped up by 2 because it's a 2-byte instruction. So the PC is now at 7DØ6. We want to jump to *loop*, which is at 7DØ2, 4 bytes back, or −4 bytes away, to use the Z80's way of thinking about it. Now, 4 in binary is ØØØØØ1ØØ and we create −4 by flipping the bits and adding 1 (2's complement, remember?). So:

```
Ø Ø Ø Ø Ø 1 Ø Ø
                    flip the bits
1 1 1 1 1 Ø 1 1
          + 1       add 1
-----------
1 1 1 1 1 1 Ø Ø
    |       |
    |       |       convert to hex
  F |       C
```

Another thing which may be worrying you: INC HL doesn't alter the flags, so I'm safe to test after the increment.

The same program with absolute jumps would have looked like:

Address		Instruction	Hex code
7DØØ		LD A, 1E	3E 1E
7DØ2	loop:	CP A, (HL)	BE
7DØ3		INC HL	23
7DØ4		JPNZ loop	C2 Ø2 7D

Notice that the JPNZ instruction has 3 bytes because it contains a whole 16-bit address; and don't forget about swapping the 2 bytes of that address around!

There's one very powerful instruction in the Jump group I haven't mentioned yet—DJNZ. It decrements the B-register by 1 and jumps (relative) only if the result is non-zero.

Suppose our little "search for 1E" program is only to search a region one hundred (hex 64) bytes long, after which it should leave the loop whether it's found a 1E or not:

```
        LD B, 64        Ø6, 64
        LD A, 1E        3E 1E
loop:   CP A, (HL)      BE
        JPZ gotcha      CA — — (address for gotcha)
        INC HL          23
        DJNZ loop       1Ø F9
```

The loop is executed one hundred times, unless a 1E is found, in which case a branch to *gotcha* occurs. In other words, DJNZ acts like a simple FOR loop in BASIC.

Note that with *all* the relative jump commands JR, JRC, JRNC, JRNZ, and JRZ, the size of jump is calculated the same way. A table of 2's complement hex codes is given in Appendix 1 for hand-coding of jumps; our utility routine HELPA in Appendix 7 will work out relative jumps automatically for you, during the writing of the code.

ADC and SBC

These are the "ADD with Carry" and "SUB with Carry" instructions. I said earlier that there is a Carry flag in the flags register. This gets set if there is a carry generated out of a register by an arithmetic instruction. The ADC instruction will act just like ADD, except that it will add 1 more in if the Carry bit has been set by a previous operation. The SBC instruction works the same way, except that it will subtract the Carry flag.

THE SHIFTS

The shift instructions, SLA, SRA and SRL, all have the effect of shifting bit-patterns around.

SLA shifts the pattern left by 1 bit, so if the B register contains:

Ø Ø 1 Ø 1 1 Ø Ø

and SLA B is executed, the result is:

Ø 1 Ø 1 1 Ø Ø Ø

(Notice that a zero is used to fill on the right.)

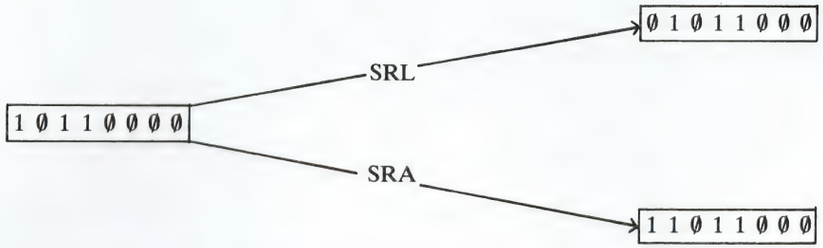
Since ØØ1Ø11ØØ = 44 and Ø1Ø11ØØØ = 88 (decimal) you can see that the effect is to multiply by 2.

Another SLA B will give:

1 Ø 1 1 Ø Ø Ø Ø

Since the senior bit is now 1, this will be seen as a negative number, and the Sign flag will be set. So far as the programmer is concerned, what's happened is that the value (176) can't be held in a byte, so we've got an overflow condition.

Right shifts work much the same way, but there's one important thing to note: SRL fills the senior bit with a zero, but SRA fills with whatever was there before. For an example of this see overpage.



The reason is this: SRL is a *shift right logical*, which simply shifts the bit pattern without altering it. SRA is a *shift right arithmetic*, which treats the operation as "divide by 2". Now, when a negative number is divided by 2 the result should still be negative, so we have to preserve the sign bit.

PUSH and POP

You'll probably remember these terms from our discussion on stacks. They're used here in exactly the same way, and allow us to access the machine stack other than through a subroutine call.

This can be useful for saving values temporarily. For instance, suppose you've got a value in BC which you want later, but just now you'd like to use BC for something else. You can write:

```
PUSH BC
.....
Code
using
BC
.....
POP BC
```

This is often done before a subroutine CALL as well, so that it doesn't matter what registers the subroutine uses: it can't interfere with the calling program's data. You may see code like:

```
PUSH BC   ┐
PUSH DE   ├─ save the registers
PUSH HL   ┘
CALL 4FA1
POP HL    ┐
POP DE    ├─ restore register values
POP BC    ┘ (note the order!)
```

assuming that the A-register is manipulated by the routine, so we don't need to save it.

Warning

Unless you deliberately choose to alter it, the stack pointer SP will be set according to the operating system of the Spectrum. There's no harm in leaving it at that value, provided you make sure that PUSHes and POPs cancel out in pairs, so that SP returns to its initial

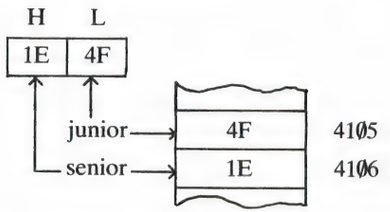
value on leaving the Machine Code routine. Similarly CALLs and RETs have to match. (USR generates a CALL, matched by the final RET that is tacked on to the end by the LOADER routine.)

A 16-BIT QUIRK

One feature of the 16-bit operations (PUSH, POP, LD in particular) which is important to grasp is the order in which bytes are transferred from register to memory and vice versa. It's like this:

```
LD (41Ø5), HL
```

will have the following effect, if HL contains 1E4F:



In other words, the least significant or "junior" byte in the register is loaded into the specified address; and the most significant or "senior" byte is loaded into the byte following this. Conversely,

```
LD HL, (41Ø5)
```

would have exactly the reverse effect. (NB: it codes as 2A Ø5 41, following the standard convention!) Similarly

```
LD HL, 1ØØØ
```

(an attempt to load HL with the value 1ØØØ hex) encodes as

```
21 ØØ 1Ø
```

so that, even though 1ØØØ is data, not an address, its bytes get transposed as usual.

CRASHES

When a BASIC program crashes, you can always BREAK out, one way or another, without losing the program. Machine Code crashes are more infuriating. Either they wipe out everything in memory (like NEW) or the machine gets totally stuck and has to be reset by pulling out the power plug for a few seconds—with the same result. You want to see a crash? Here's the simplest one I know that doesn't NEW:

```
RANDOMIZE USR 996
```

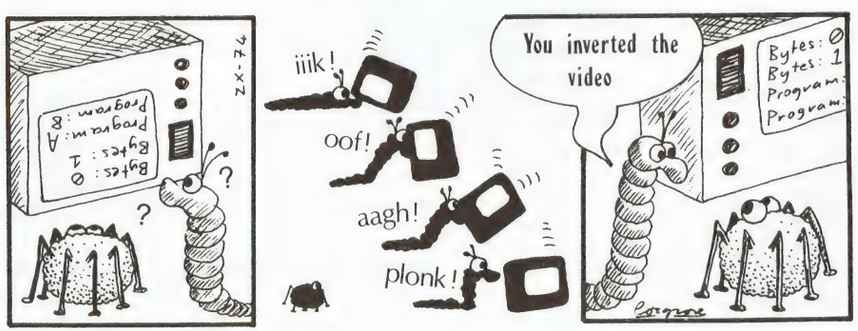
Noises; colour; and no keyboard response. Yugh. RANDOMIZE USR 14ØØ is even bleaker a prospect.

(The old steam-driven ZX81 used to produce fantastic op-art crashes: the Spectrum is better protected and on the whole just gets stuck in a fairly civilized fashion, or takes its ball away and refuses to play at all.)

Crashes are unavoidable in Machine Code work—as you'll learn pretty quickly. Pulling out the plug is the only certain cure; and then you'll have to start again. But there are a few precautions worth taking to reduce the chances:

1. Check all Machine Code listings scrupulously and make sure you have input them correctly.

2. *Don't* use HALT (Opcode 76 hex). It's *not* like the BASIC STOP command—it just causes a crash.
3. Make sure all CALLs and RETs match, as do PUSHes and POPs.
4. Call the *correct* starting address.
5. Unless there's not much to lose, SAVE what you can on tape before trying the routine out for the first time.



Machine code has no instruction for multiplying numbers; but you can do it if you combine arithmetic, logic, and shifts. Digging deeper now . . .

12 A Machine Code Multiplier

Now let's write a few simple routines. Remember I said that there's no Z80 multiply instruction? Let's write a subroutine to do the job.

AN EXAMPLE

First we should examine the nature of the problem, and there's no better way of doing that than looking at an example. We'll keep things as simple as possible, and work in 8-bit registers; so if we want to multiply 9 by 13 it will look like:

```
    0 0 0 0 1 0 0 1
×   0 0 0 0 1 1 0 1
```

Now we can treat this as a conventional long multiplication, but because it's in binary, it's actually easier than usual; if the current digit we're multiplying by is 1, copy the top line; if it's zero, do nothing:

```
    0 0 0 0 1 0 0 1      P
×   0 0 0 0 1 1 0 1      Q
-----------------
    0 0 0 0 1 0 0 1
    0 0 1 0 0 1 0 0
    0 1 0 0 1 0 0 0
-----------------
    0 1 1 1 0 1 0 1
```

Of course we've had to add in zeros on the right at each stage, just as we would in a decimal long multiplication. In machine code terms, that's equivalent to a shift left. I've called the two numbers P and Q, for reference.

While P is shifted left, it's also going to be convenient to shift Q right, because that way we only need to keep examining the junior bit of Q to determine whether to add P into the sum or not.

PROCEDURE

Assume that P and Q are in the D and E registers. The procedure is:

1. Set the A-register to zero.

2. If the junior bit of E is 1 then add D into A.
3. Shift D left.
4. Shift E right.

repeat these steps 8 times

Here's a first stab at the code:

```
LD A, 00
LD B, 08
```

The first step's obvious; the second sets B to act as a loop counter in conjunction with a DJNZ to come at the end. Now we want to test the junior bit of E. The only way we currently have of doing that is to use a mask pattern (00000001) with an AND operation, so let's set up the C register to that pattern:

```
LD C, 01        [see below for hex coding]
```

We can only AND with the A-register, which will destroy its current contents, so we'll save it in L first:

```
loop: LD L, A
```

then extract the junior bit of E, and restore the A-register:

```
LD A, C
AND A, E
LD A, L
```

If the result of the AND was zero, we need to jump round the "add D into A" part of step 2 so:

```
JRZ shift
```

(Note that since LD doesn't affect the flags, the JRZ still refers to the AND.) Otherwise perform the ADD:

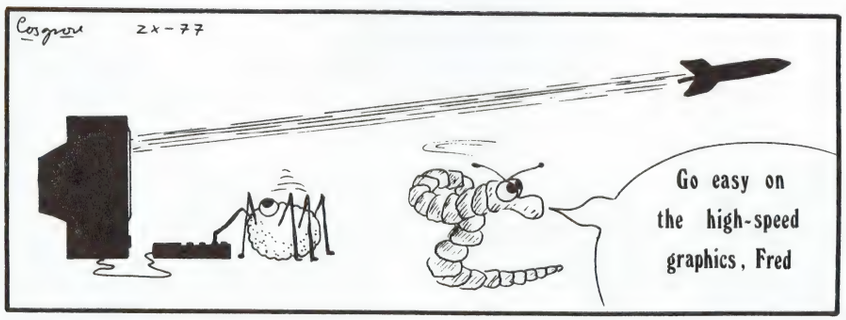
```
ADD A, D
```

Now do the shifts:

```
shift: SLA D
       SRA E
```

and see if we've done the loop enough times yet:

```
DJNZ loop
(RET)
```



THE CODE

Here's the whole thing:

```
        LD A, 00        3E 00
        LD B, 08        06 08
        LD C, 01        0E 01
loop:   LD L, A         6F
        LD A, C         79
        AND A, E        A3
        LD A, L         7D
        JRZ shift       28 01
        ADD A, D        82
shift:  SLA D           CB 22
        SRA E           CB 2B
        DJNZ loop       10 F3
```

If you want to try this program out, you'll have to arrange for the D and E registers to hold the values to be multiplied. So you could precede the program by something like:

```
LD HL, 7D00     21 00 7D
LD D, (HL)      56
INC HL          23
LD E, (HL)      5E
```

and then POKE 7D00 (hex) and 7D01 (hex) with the values to be multiplied, before calling the program. These two bytes will, of course, be the two zero bytes at the beginning of the routine, so the LD HL, 7D00 will start in 7D02. Note that I didn't assign actual addresses to the program. This is because all the jumps are relative, so actual addresses are unimportant; only displacements matter.

You'll also need to *output* the answer: at the moment, it's just sitting in the A-register. One way is to reserve yet another data byte 7D02 for the answer, as I did in the adding program earlier: this means you must append the code:

```
LD (7D02), A    32 02 7D
```

and use 3 data bytes when loading the code in. To get the answer, use

```
PRINT PEEK 32002
```

Alternatively, you can transfer the result from the A-register to the C-register, using

```
LD B, 0         06 00
LD C, A         4F
```

and call the Machine Code with

```
PRINT USR 32002
```

Recall that USR is a function, holding the value of the BC-register on returning to BASIC; so this command both runs the Machine Code and prints out the answer! (I've assumed we're back to only two data bytes, which is why we start at 32002.)

BIT

Now, I have a confession to make; there's an easier way of testing to see if the junior bit of E contains 1. There's an instruction BIT 0, E which does the job. So:

```
loop:  LD L, A      6F
       LD A, C      79
       AND A, E     A3
```

becomes just:

```
loop:  BIT 0, E     CB 43
```

and the LD A, L has to disappear as well.

Why didn't I tell you that in the first place? Well, firstly, I promised to use only the subset of instructions in the table, a promise I've now broken. But I've made an important point in the process—that it's possible to do things satisfactorily without knowing the full instruction set.

This has been something of an academic example; I chose it because it uses several common instructions in conventional, but not necessarily obvious, ways, but I'm not suggesting that you will find a need for dozens of 8-bit integer multiplications.

A large part of the Spectrum's memory is devoted to a single task: maintaining the display on the TV screen. To make use of the display in a Machine Code program, it is necessary to understand how the information is arranged.

13 The Screen Display

The information that controls the screen display is held in two segments of RAM, known as the *display file* and the *attributes file*. The first controls the characters and hi-res graphics, the second controls colours and things like FLASH and BRIGHT. They are organized in rather different ways. The simpler of the two is the attributes file, and I'll start with that.

ATTRIBUTES FILE

This occupies 24 * 32 = 768 bytes in the following locations:

	Decimal	Hex
Start of attributes file	22528	5800
End of attributes file	23295	5AFF

The first 22 * 32 = 704 handle the usual PRINT area of the screen, rows 0–21 for PRINT AT commands. The last 2 * 32 = 64 handle the lower two-line region usually used for error messages and editing. The PRINT region ends at address 23231 decimal, 5ABF hex; the lower section of screen starts at 23232 (5AC0).

Numbering the rows as usual from 0 to 21 (and thinking of the lower regions as extra lines 22 and 23) and the columns from 0 to 31, then row r, column c corresponds to

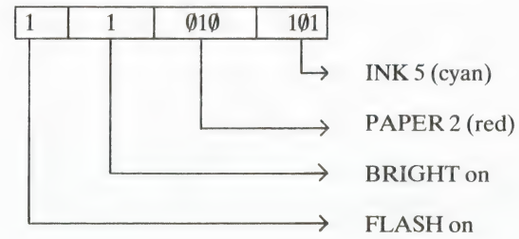
22528 + 32 * r + c

and this address holds the *attribute* for that position.

Each attribute is a single byte, whose eight bits are split up as follows:

FLASH	BRIGHT	three		PAPER	three		INK
			for			for	
on/off	on/off	bits		colour	bits		colour

with 0 standing for "off" and 1 for "on". So the byte 11010101 breaks down like this



(and pretty awful it will look with that combination, too). If you wanted this attribute to apply to the position in row 5, column 7, you would have to store the above byte in location

22528 + 32 * 5 + 7

which is

22695.

The attribute itself is 213 in decimal; so the command

POKE 22695, 213

will store the byte in the right place. Of course you won't see the ink colour if there's only a blank space to be printed, so you should also enter something like

PRINT AT 5, 7; "*"

to make sure this works.

Because there are 32 columns to each row, the address for the space immediately below a given one is found by adding 32 decimal, 20 hex. So the attributes are laid out like this:

top row	5800	5801	5802	...	581D	581E	581F	PRINT area
1st row	5820	5821	5822	...	583D	583E	583F	
2nd row	5840	5841	5842	...	585D	585E	585F	
.	.	.	.	...	.	.	.	
.	.	.	.	...	.	.	.	
.	.	.	.	...	.	.	.	
21st row	5AA0	5AA1	5AA2	...	5ABD	5ABE	5ABF	
22nd row	5AC0	5AC1	5AC2	...	5ADD	5ADE	5ADF	Messages
23rd row	5AE0	5AE1	5AE2	...	5AFD	5AFE	5AFF	

DISPLAY FILE

This is much more complicated, because each character in the display is stored as a list of *eight* bytes, each corresponding to one row of the 8 × 8 array of hi-res pixels that the character occupies. To make life worse, these are *not* stored in the obvious order.

When you load a picture into the Spectrum using LOAD SCREEN$ you must have noticed the curious order in which the picture is "painted in". In fact this is precisely the order of the bytes in the display file. The easiest way to see this is to fill the screen with black ink, SAVE, and LOAD back:

```
10 FOR r = 0 TO 21
20 PRINT "[32 inverse spaces]"
30 NEXT r
40 SAVE "blank" SCREEN$
```

Get this on tape; now enter

CLS: LOAD "blank" SCREEN$

and watch how the screen blacks in.

Now let's get more numerical. The addresses are:

	Decimal	Hex
Start of display file	16384	4000
End of display file	22527	57FF

and the total number of bytes is 32 * 24 * 8 = 6144 decimal, 1800 hex.

The best way to think of the display file is to divide the screen up into three blocks:

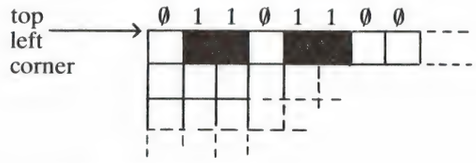
BLOCK 1	Rows 0–7 of the screen
BLOCK 2	Rows 8–15 of the screen
BLOCK 3	Rows 16–23 of the screen

Note that Block 3 includes the "message" rows 22 and 23. Each block requires 6144/3 = 2048 bytes (800 hex). The first 2048 bytes of the display file handle Block 1, the next handle Block 2, the last handle Block 3; and the ordering arrangement is the same in each block. In hex we have

Block	Start address	End address
1	4000	47FF
2	4800	4FFF
3	5000	57FF

To understand the arrangement within a block, I'll deal with Block1: the others are similar (but remember that Block 3 includes the "messages" area). The process is easier to describe on the hi-res coordinate grid, with 256 × 176 pixels. Number the rows from 0 to 175 and the columns from 0 to 256, with row 175 at the top and column 0 at the left, as usual for PLOT *column, row* commands. Then Block 1 is made up of rows 175–112, a total of 8 × 8 = 64 rows, each of 256 pixels.

The first 32 bytes of the display file hold the information for row 175. Each byte governs an 8-column segment. The first byte deals with columns 0–7, using one bit per pixel. For instance, if address 4000 holds the byte 01101100 then the first eight pixels at the top left of the screen are



with spaces for "0" and black squares for "1".

The next byte deals with columns 8–15, the next with 16–23, and so on until after 32 bytes the whole of row 175 is accounted for.

To see this in operation, CLS and then do

POKE 16384, BIN 01101100

and you'll see two short dashes in the top row (the 11's in the binary). Try

POKE 16384, BIN 10101010

and get four dots. Now change the 16384 to 16385, 16386, . . . up to 16415, and fill up row 175.

The next byte is in address 16416. Try

POKE 16416, BIN 10101010

and you'll see that this is *not* the first eight columns of the next row, row 174! It is in fact the first eight columns of row 167. Note that 175 − 8 = 167: it skips on *eight* hi-res rows. This byte, and the 31 that follow it, fill out the rest of row 167. Then the machine moves on to row 167 −8 = 159, and continues doing this 8-row skip until it gets to the bottom of Block 1. So, in 32-byte chunks, the rows come in order

175 167 159 151 143 135 127 119

If it were to drop by eight again we'd be in row 111, which is no longer in Block 1. (It's the top row of Block 2.) So instead the Spectrum jumps up to the rows immediately

below the ones just dealt with; and the next eight 32-byte sections deal with rows

174 166 158 150 142 134 126 118

Then the lines below these:

173 165 157 149 141 133 125 117

and so on until eventually the bottom row of the block is reached:

168 160 152 144 136 128 120 112

That's Block 1 completed. The next 2048 bytes handle Block 2 similarly, and finally Block 3 is dealt with. The order within each block is exactly the same: the addresses shift up by 2048 and the positions shift down 64 rows on the screen, from one Block to the next.

This looks pretty complicated; and the best way to handle it is to think first Block by Block, then in 8-row chunks, and only then within a given row. The arrangement makes more sense if we recall that a character occupies an 8 × 8 pixel square (in hi-res); that is, one PRINT AT position. Then the eight rows of the character are represented by the corresponding eight bytes (exactly as for user-defined graphics, see *Easy Programming**, page 49). So a display of characters in Block 1 is arranged in the display file like this:

32 bytes for the top row of each character in line 0
32 " " " " " " 1
. . .
32 " " " " " " 7

and then

32 bytes for the *second* row of each character in line 0
32 " " " " " " 1
. . .
32 " " " " " " 7

Then on to the third, fourth, . . ., eighth rows. It still looks messy; but not perhaps quite so bad as it did at first sight.

I'll get us used to the attributes file first, in the next chapter; and then we'll take a stab at the display file in the one after that.

* *Easy Programming for the ZX Spectrum* by Ian Stewart and Robin Jones, Shiva.

Fill the screen instantly with one (or more) colours; turn off all flashing pixels; scroll a single column of colours . . .

14 The Attributes File

The simplest place to start using the attributes file is to write a Machine Code routine to print a square of a chosen colour at the top left corner (the start of the file, address 5800 hex). I'll use one data byte (7D00 as usual) to hold the colour (attribute byte) and the code is then:

LD A, attribute	3A 00 7D
LD (5800), A	32 00 58

Load this in, STOP; and set the data by

POKE 32000, 32

(Since 32 = 4 * 8 this is the PAPER 4 attribute.) Now GO TO 300 and input "r" to run . . .

. . . Good, it works! Here are a few exercises to test your grasp so far: answers at the end of the chapter.

1. Make it a red square, not green.
2. Make it a magenta square.
3. Make it blue and FLASHing.
4. Make it yellow and BRIGHT.
5. Put it in row 0, column 1.
6. Put it in row 3, column 7.

For 1–4 you just need to POKE 32000 suitably; for 5 and 6 you must modify the address 5800 to the correct place in the attributes file.

Next, let's fill in the whole top row in green. We'll need a loop, using a counter that is set to 32 and decrements on each pass: we test to see if it's zero, and jump if not.

	LD HL, a-file	21 00 58
	LD B, 32 dec	06 20
loop:	LD (HL), 32 dec	36 20
	INC HL	23
	DEC B	05
	CP B	B8
	JRNZ loop	20 F9

No data bytes this time, so go straight to "r" when the option comes up.

We can easily get two, three, . . . rows by making the counter B larger. If we start with LD B, 64 dec [06 40] we get *two* rows going green; [06 60] gives three; and so on up to [06 E0] which gives seven. Adding 20 hex (32 dec) more to B means we start at 00; the first decrement of B takes this to 255 (no carry digits remembered!) so it's just as if we'd started at 256 for B. And indeed, if you change the second line above to

LD B, 0	06 00

then the top *eight* rows fill in with green. Beyond this we can't go, at least not by changing the values loaded into the B-register, because that's a 1-byte register. Before we get round that, let's be a trifle more efficient. Using B as a counter is *automatic* with the DJNZ command, and that saves a few bytes. The job is done equally well by

```
        LD HL, a-file       21 00 58
        LD B, 0             06 00
loop:   LD (HL), 32 dec     36 20
        INC HL              23
        DJNZ  loop          10 FB
```

Try it, and see.

INSTANT COLOUR

Now, to get all 24 rows of screen changing colour, we can either use BC as a counter (starting it at 0300) and add in a CP C and another JRNZ *loop* (to see if BC is zero you have to check B and C separately). Or we could loop the above using another register as a counter. Or, totally unsophisticated as we are, we could just jam three copies of the program end to end, starting in different parts of the attributes file.

Let's do that first . . .

```
        LD HL, 5800         21 00 58
        LD B, 0             06 00
loop 1: LD (HL), 32 dec     36 20
        INC HL              23
        DJNZ loop 1         10 FB
        LD HL, 5900         21 00 59
        LD B, 0             06 00
loop 2: LD (HL), 32 dec     36 20
        INC HL              23
        DJNZ loop 2         10 FB
        LD HL, 5A00         21 00 5A
        LD B, 0             06 00
loop 3: LD (HL), 32 dec     36 20
        INC HL              23
        DJNZ loop 3         10 FB
```

Run this, with 0 data bytes: the whole screen fills instantly. Including the "messages" area. To avoid the latter, change the third "LD B, 0" to "LD B, 192 dec" or [06 C0] to get only six rows on the third pass.

Incidentally, did you notice that it is not necessary to reset B to zero each time? That's what it is already! So those lines could be deleted (except the first, which is always necessary; and the third if you want only 22 lines' worth to go green).

Obviously there must be a quicker way; but the above has one nice feature: we can change the colours as we go. If you make the second occurrence of [36 20] become [36 10] and the third [36 30] (most easily done by POKE 32016, 16: POKE 32026, 48) you get three blocks of colour:

GREEN
RED
YELLOW

and of course changing the attribute bytes produces other, similar, effects.

No excuses: let's get that extra loop working properly. We'll use D as a loop-counter. For a change, we'll go for a magenta screen.

```
        LD HL, a-file       21 00 58
        LD D, 03            16 03
outer:  LD B, 0             06 00
loop:   LD (HL), 24 dec     36 18
        INC HL              23
        DJNZ  loop          10 FB
        DEC D               15
        CP D                BA
        JRNZ  outer         20 F5
```

Notice that we do not increment HL in the outer loop, because that is happily ticking its way up through the attributes file already: it is the failure of B alone to act as a counter that must be overcome.

To see how much slower this routine is in BASIC, see *Further Programming*, Chapter 7.

DEFLASH AND REFLASH

The next routine runs through the attributes file and turns off all FLASHes, otherwise leaving the attributes unchanged. Now the FLASH bit in the attribute is the one at the left-hand end; that is, the most senior bit. What we must do is set it to 0 while leaving the rest unchanged.

One way to do this is to use a *mask* pattern of bits: if we put the attribute byte into the A-register and then AND it with the bit-pattern 01111111, then the first bit goes to 0 and the others will be unchanged. Using this idea, the code is:

```
        LD BC, 768 dec      01 00 03
        LD HL, a-file       21 00 58
loop:   LD A, (HL)          7E
        AND 127 dec         E6 7F
        LD (HL), A          77
        INC HL              23
        DEC BC              0B
        LD A, 0             3E 00
        CP B                B8
        JRNZ  loop          20 F5
        CP C                B9
        JRNZ  loop          20 F2
```

Note that BC is used as a loop-counter: 768 is of course the length of the attributes file. Both B and C must be tested for zero to end the loop.

Load this in, with 0 data bytes; then STOP. We need something to test it on. Add program lines

```
1000  CLS: FOR i = 1 TO 704: PRINT FLASH
      (INT (2 * RND) ); CHR$ (65 + i - 20 * INT (i/20) ); :
      NEXT i
1010  GO TO 300
```

Use GO TO 1000. The screen will fill with letters, some flashing. Input "r" for the option, and the flashes turn off.

Try setting INK and PAPER to various colours, and repeat: note that the colours don't change.

The converse problem is to turn the whole screen to FLASH. Instead of using AND 127 dec as a mask-pattern, we have to use OR 128 dec (128 is 10000000 in binary; OR makes the first bit go to 1 and leaves the others unchanged). So we need the identical program, except that the line

```
AND 127 dec          E6 7F
```

becomes

```
OR 128 dec           F6 80
```

You can use line 1000 to test, as before. What happens if you use XOR with 128?

SET AND RES

There is a more direct way to achieve the above changes in the bit-pattern. The command SET will *set* a given bit in a given register to 1; and RES will *reset* it (to 0).

Bits within the byte are numbered in this order:

7	6	5	4	3	2	1	0

so that the larger numbers refer to more senior bits. A command like

```
SET 4, D
```

makes bit 4 of the D-register become 1; and

```
RES 6, E
```

makes bit 6 of the E-register become zero.

So instead of using AND 127 dec we could have used

```
RES 7, A             CB BF
```

and instead of OR 128 dec

```
SET 7, A             CB FF
```

These occupy the same number of bytes so there is no need to adjust the relative jumps: make the changes and check the routines still work.

COLUMN SCROLL

Now let's write a routine to scroll a column of attributes—for the moment, column 31, far right. Remember that the attributes of the square immediately below a given one live at an address 32 higher. So we ought to use indexing, with a 32 displacement. The idea is

to transfer the attributes of this lower square into the D-register, and then transfer them out into the upper square; loop this 21 times for the whole column. Which leads to:

```
        LD BC, 32 dec        01 20 00
        LD IX, start         DD 21 1F 58
        LD A, 21 dec         3E 15
loop:   LD D, (IX + 32)      DD 56 20
        LD (IX), D           DD 72 00
        ADD IX, BC           DD 09
        DEC A                3D
        CP B                 B8
        JRNZ loop            20 F4
```

Note that the entry 1F in the second command is the column number 31 that we intend to scroll, but written in hex. For a different column, change this number.

Again we need something to test this on. So load it in, STOP, and add

```
1000  FOR e = 0 TO 21: PRINT PAPER e - 7 * INT (e/7);
      " [32 spaces] "; : NEXT e
1010  GO TO 300
```

Now GO TO 1000 for a nice stripey pattern to test with; and then enter option "r" to run.

You'll see column 31 scroll up one space. For more spaces, loop the routine in BASIC. For a different column, change the 1F.

You'll also notice that the bottom attribute in the column is left unchanged. To make it into a white square (attribute 56 = 7 * 8) add the following line to the Machine Code, after the JRNZ:

```
LD (IX), 56 dec      DD 36 00 38
```

As an exercise for you: think how to scroll a complete block of columns (say columns 5 to 17 inclusive) by looping this routine within Machine Code, and incrementing the start address for the IX each time.

PATTERN-GENERATOR

Finally, here's a routine that jazzes up INSTANT COLOUR to give fancy patterns. I'll leave it to you to work out just what it's doing: there are 0 data bytes.

```
LD B, 0          06 00
LD D, 0          16 00
LD HL, 5800      21 00 58
LD (HL), D       72
INC HL           23
LD A, 7          3E 07
ADD A, D         82
LD D, A          57
DJNZ ?           10 F8
LD HL, 5900      21 00 59
LD (HL), D       72
```

```
INC HL          23
LD A, 7         3E Ø7
ADD A, D        82
LD D, A         57
DJNZ ?          1Ø F8
LD HL, 5AØØ     21 ØØ 5A
LD (HL), D      72
INC HL          23
LD A, 7         3E Ø7
ADD A, D        82
LD D, A         57
DJNZ ?          1Ø F8
```

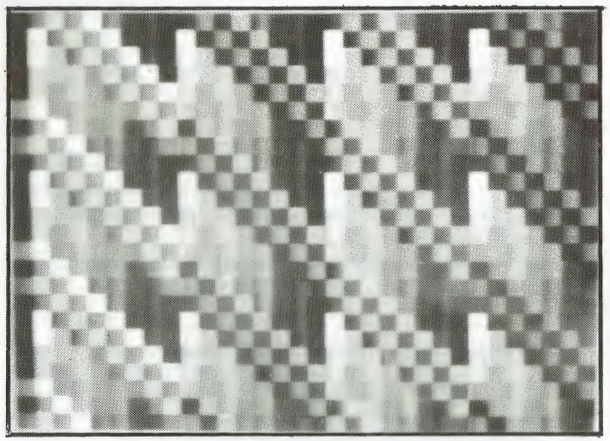


Figure 14.1 LD A, 31 for this chequered pattern. What do the other values of A give? There's 255 to try.

For different patterns, change the lines LD A, 7 to load other numbers into A: for example LD A, 8 or LD A, 32 or LD A, 31 (decimal!). Any 2-digit hex number can be used instead of the Ø7; and different numbers can be used on each of the three occurrences.

To clear the screen first, stop (using option "a"), then CLS, then GO TO 3ØØ, then use option "r".

Can you work out what it's doing? And how?

ANSWERS

1. POKE 32ØØØ, 16.
2. POKE 32ØØØ, 24.
3. POKE 32ØØØ, 136.
4. POKE 32ØØØ, 112.
5. Change 58ØØ to 58Ø1 [3A ØØ 7D 32 Ø1 58].
6. Change 58ØØ to 5867 [3A ØØ 7D 32 67 58].

Its complicated structure makes the display file a little harder to use; but the results amply reward persistence.

15 The Display File

The main thing to remember is that the display file starts at 4ØØØ hex, is 18ØØ bytes (hex) long; and naturally divides into three blocks, running from 4ØØØ–47FF, 48ØØ–4FFF, 5ØØØ–57FF respectively.

The best way to get a feel for it is to experiment, making changes to the display file section of memory, and seeing what the result is.

For example, suppose you put an identical value in each address. What happens?

For definiteness, take the byte 1Ø1Ø111Ø (binary), or AE (hex), or 174 (dec). We can use HL to point to the address in the display file, and BC as a loop-counter. So we have:

```
        LD HL, d-file       21 ØØ 4Ø
        LD BC, 18ØØ         Ø1 ØØ 18
loop:   LD (HL), 174 dec    36 AE
        INC HL              23
        DEC BC              ØB
        CP B                B8
        JRNZ  loop          2Ø F9
        CP C                B9
        JRNZ  loop          2Ø F6
```

What you get is a pattern of vertical stripes. These are formed by the bit-pattern of the number 174 in binary, that is, 1Ø1Ø111Ø: there is a black stripe of width 1, then a white stripe of width 1, then a black stripe of width 1, then a white stripe of width 1, then a black stripe of width 3, and finally a white stripe of width 1: the whole thing repeats 32 times from left to right across the screen. This is because every row of the display gets loaded with the same bit-pattern, repeated 32 times; these align vertically to give stripes.

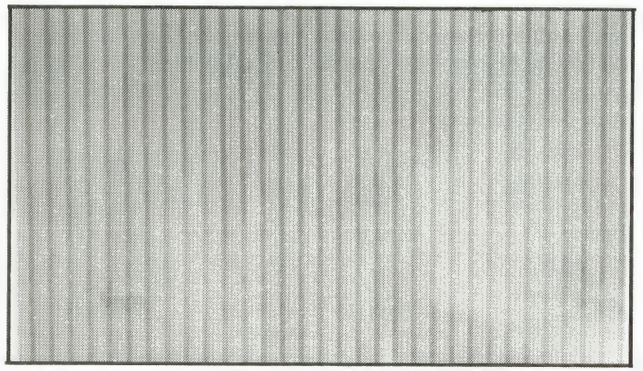


Figure 15.1 Identical bytes give repetitive stripes.

Change the 174 to other numbers, and see what patterns appear. In particular, try 1, 15, and 170, which in hex are 01, 0F, and AA.

HORIZONTAL LINES

The next routine draws horizontal lines (by putting 255 into the bytes for certain rows of display). You may wish to clear the screen before using option "r", as at the end of Chapter 14.

```
        LD A, 3                 3E 03
        LD B, 0                 06 00
        LD HL, d-file           21 00 40
loop:   LD (HL), 255 dec        36 FF
        INC HL                  23
        DJNZ  loop              10 FB
        DEC A                   3D
        CP B                    B8
        JRZ  skip               28 08
        PUSH AF                 F5
        LD A, 7                 3E 07
        ADD A, H                84
        LD H, A                 67
        POP AF                  F1
        JR  loop                18 EF
skip:   (RET instruction        C9)
```

PATTERNS

By loading different bytes into parts of the display file, you can create some quite striking patterns. This one uses the C-register, which is starting at 00 and decrementing to FF, FE, FD, . . . and down to 00 (three times over altogether) to define the byte loaded in. So, by *careful* analysis of the display, you can actually see exactly what the display file ordering is. In practice the pattern makes this analysis confusing unless you know the answer already . . .

```
        LD BC, 768 dec          01 00 18
        LD HL, d-file           21 00 40
loop:   LD (HL), C              71
        INC HL                  23
        DEC BC                  0B
        CP B                    B8
        JRNZ  loop              20 FA
        CP C                    B9
        JRNZ  loop              20 F7
```


As a variation on the theme, try changing the *loop* line to load the B-register in instead:

```
loop:   LD (HL), B              70
```

This gives a different pattern. It takes 256 moves round the loop to change the value of B (being the senior byte of the loop-counter), corresponding to eight hi-res rows of the screen. The pattern makes it clear that the top eight rows each have different values of B, so that the first 256 bytes do *not* correspond to rows 175–168 (as they would if the rows were dealt with in order from top to bottom); and the way the pattern repeats within an 8-row block shows that the first 256 bytes actually hold lines 175, 167, 159, etc. as I explained above. The three-block structure is very clear in this pattern, as well as the previous one.

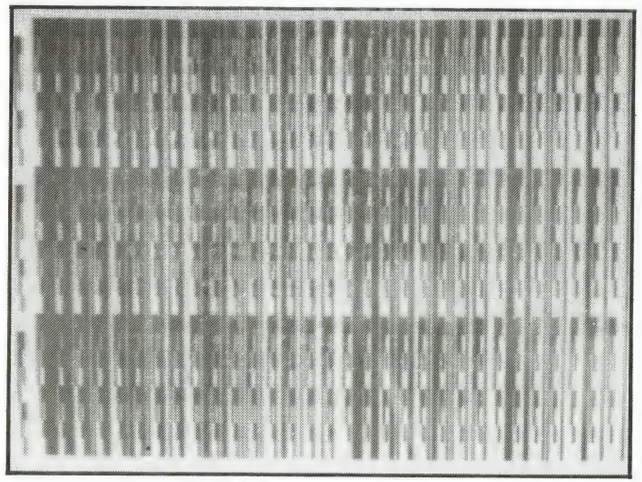


Figure 15.2 Variable bytes give strange patterns. See the three blocks of screen? Can you spot the binary numbers responsible?

Routines like this give us some confidence that we *can* make the display file do interesting things, if we really want it to; but they're not exactly *useful* except as a confidence-builder. So now let's move towards something with a definite purpose.

CROSS-HATCH

This routine has the advantage that the detailed structure of the display file is not important. It looks through the file and replaces every zero byte 00000000 by the byte 10101010 (AA hex). The effect is to hatch in the blank areas of screen with fine vertical pinstripes.

```
        LD HL, d-file           21 00 40
        LD BC, 768 dec          01 00 18
        LD A, 0                 3E 00
loop:   CP A, (HL)              BE
        JRNZ skip               20 02
        LD (HL), AA hex         36 AA
```

```
skip:  INC HL          23
       DEC BC          ØB
       CP B            B8
       JRNZ loop       2Ø F6
       CP C            B9
       JRNZ loop       2Ø F3
```

The command LD A, Ø isn't really necessary: A holds zero as a matter of course unless it is loaded with something else. But it makes the way the program works look a bit clearer. Check that it isn't needed by deleting it. (The quickest way to do this is to use POKE 32ØØ6, Ø which converts the line to ØØ ØØ. The code ØØ means NOP—No OPeration—and does nothing except waste time. This makes it ideal for experimentally deleting a command, because the other hex codes don't have to be moved around in memory.)

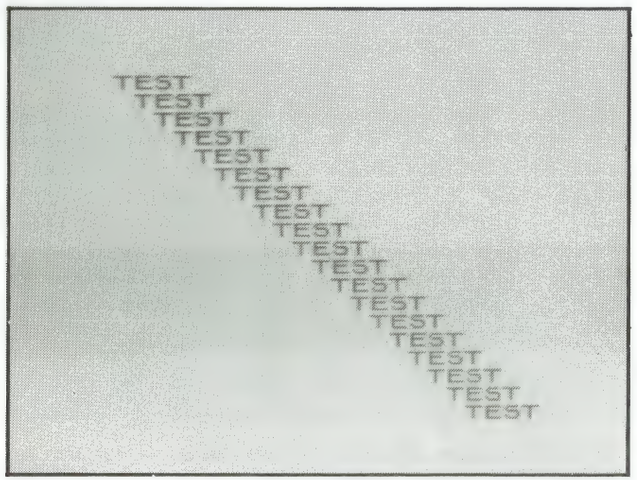


Figure 15.3 Before doing Cross-hatch . . .

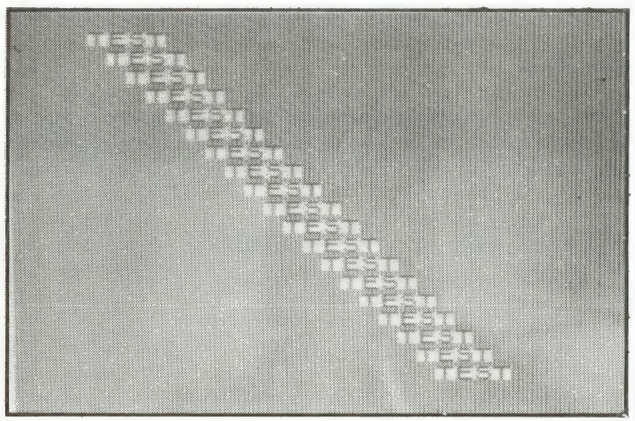
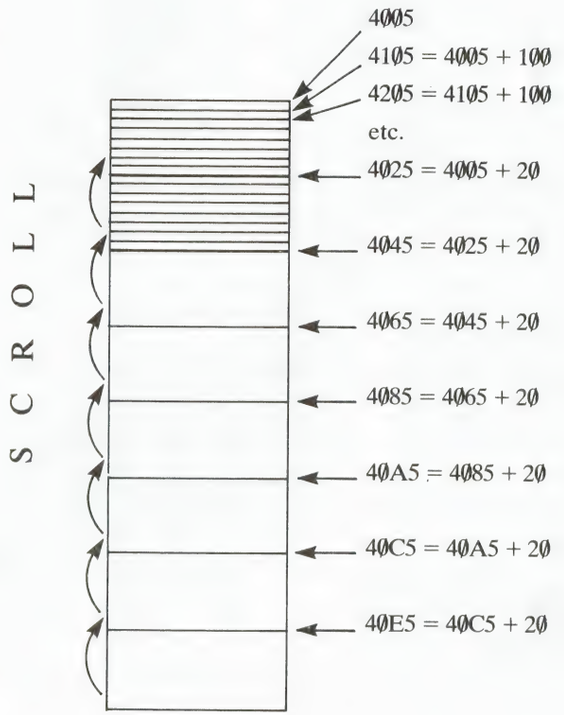


Figure 15.4 . . . and after.

SINGLE-COLUMN SCROLL

Now for something more complicated: scrolling a single column of the screen. To simplify the problem a little, I'll only scroll within one of the three Blocks; that is, scroll an 8-row section of the column. For definiteness, I'll work in Block 1, column 5.

Column 5 looks like this: it comes in 8-line sections, each corresponding to one character (and to one PRINT AT cell), and each line is stored at the address shown.



To avoid confusion in terminology, I'll call each 8-line square a *row* (as it is for a PRINT AT statement) and each individual line a *line*. Then the top line of row Ø is contained in address 4ØØ5 (hex) in the display file. Adding 32 decimal (2Ø hex) gives the top line of row 1; and so on down to row 7.

To get the second line in row Ø we must add 256 (dec) or 1ØØ (hex) to the original 4ØØ5, giving

4ØØ5 + Ø1ØØ = 41Ø5

Continuing to add successive 2Ø (hex)'s we get the second lines of the other seven rows. Then we go on to the third lines, . . . and eventually finish on the eighth lines.

That's the structure of the column (Block 1). To scroll up any byte by one row, we move it to the address 32 places earlier (decimal). We want to lose the top row altogether, and leave the seventh row blank at the end.

This is very like the column scroll of attributes that I wrote earlier, except that now it has to be looped eight times to move all eight lines in a square. (A routine to scroll just the top 1/8 of each character wouldn't be much use—especially as they're all blanks!)

So we need to use indexing, with a displacement of Ø or 32, as before.

In the faint hope of making all this clearer, let me first give a BASIC version of the routine.

```
2000  LET ix = 256 * 64 + 5
2010  FOR l = 1 TO 8
2020  FOR a = 1 TO 7
2030  LET b = PEEK (ix + 32)
2040  POKE ix, b
2050  LET ix = ix + 32
2060  NEXT a
2070  POKE ix, 0
2080  LET ix = ix + 32
2090  NEXT l
```

Here I've used lower-case variables like ix to correspond to the upper-case registers like IX. The a-loop shifts up all the top lines; line 2070 pokes in the blank for the seventh row; and the *l*-loop repeats the process for the 2nd, 3rd, . . ., 8th lines in each character.

Set up something to test this on using LIST 2000: GO TO 2000.

If you try this, it works—but it's rather slow: the characters visibly "trickle" upwards. But at least we're on the right track. (This is a useful debugging trick: write a BASIC version of the routine first, to check the idea is sensible; debug *that*; then move into Machine Code for the real McCoy.)

Turning the BASIC into Machine Code gives:

```
        LD DE, 0020      11 20 00
        LD IX, 4005      DD 21 05 40
        LD L, 08         2E 08
loop 2: LD A, 07         3E 07
loop 1: LD B, (IX + 20)  DD 46 20
        LD (IX), B       DD 70 00
        ADD IX, DE       DD 19
        DEC A            3D
        CP D             BA
        JRNZ  loop 1     20 F4
        LD (IX), 00      DD 36 00 00
        ADD IX, DE       DD 19
        DEC L            2D
        PUSH AF          F5
```



```
        LD A, 0          3E 00
        CP L             BD
        POP AF           F1
        JRNZ  loop 2     20 E4
```

To test this, add to the BASIC

```
1000  FOR i = 0 TO 21: FOR j = 0 TO 31:
      PRINT CHR$ (65 + i); : NEXT j: NEXT i
```

Then GO TO 1000; then run the routine (either via GO TO 300 or by a direct RANDOMIZE USR 32000). The top Block of column 5 does indeed scroll up—at a respectable pace. If you loop the Machine Code in BASIC you get an acceptable fast scroll

```
3000  FOR k = 1 TO 8: RANDOMIZE USR 32000: NEXT k
```

If you loop it in Machine Code, it's so fast you can hardly see the characters disappearing.

MULTICOLUMN SCROLL

Having got this one working, it's easy to jazz it up into a routine to scroll a whole section of columns, within a Block. There's a bit more preparatory work, though.

Set aside four data bytes in 7D00–7D03. These will hold the start column, Block, width of section to be scrolled, and a dummy zero needed for tidiness:

7D00	column	(e.g. 05 for column 5 start)
7D01	Block	(40, 48, or 50 for the three Blocks)
7D02	width	(e.g. 11 for a 17 (dec) width)
7D03	00	(dummy)

The Machine Code, which starts at 7D04 (called by RANDOMIZE USR 32004) looks like this: notice that the middle bit is just a repeat of what we developed above.

```
        LD BC, (7D02)    ED 4B 02 7D
        LD DE, 0020      11 20 00
        LD IX, (7D00)    DD 2A 00 7D
loop 3: PUSH IX          DD E5
        LD L, 08         2E 08
loop 2: LD A, 07         3E 07
loop 1: LD B, (IX + 20)  DD 46 20
        LD (IX), B       DD 70 00
        ADD IX, DE       DD 19
        DEC A            3D
        CP D             BA
        JRNZ loop 1      20 F4
        LD (IX), 00      DD 36 00 00
        ADD IX, DE       DD 19
        DEC L            2D
        PUSH AF          F5
```

```
        LD A,Ø          3E ØØ
        CP L            BD
        POP AF          F1
        JRNZ loop 2     2Ø E4
        POP IX          DD E1
        DEC C           ØD
        PUSH AF         F5
        LD A,Ø          3E ØØ
        CP C            B9
        POP AF          F1
        JRZ skip        28 Ø4
        INC IX          DD 23
        JR loop 3       18 D2
skip:   (RET            C9)
```

To use this, POKE the data bytes with the relevant numbers (e.g. 5, 64, 17, Ø—remember, POKE works in decimal, not hex!). Then use GO TO 1ØØØ with the little test program above, to set up something to scroll. Then either GO TO 3ØØ with option "r", or RANDOMIZE USR 32ØØ4. Up she goes!

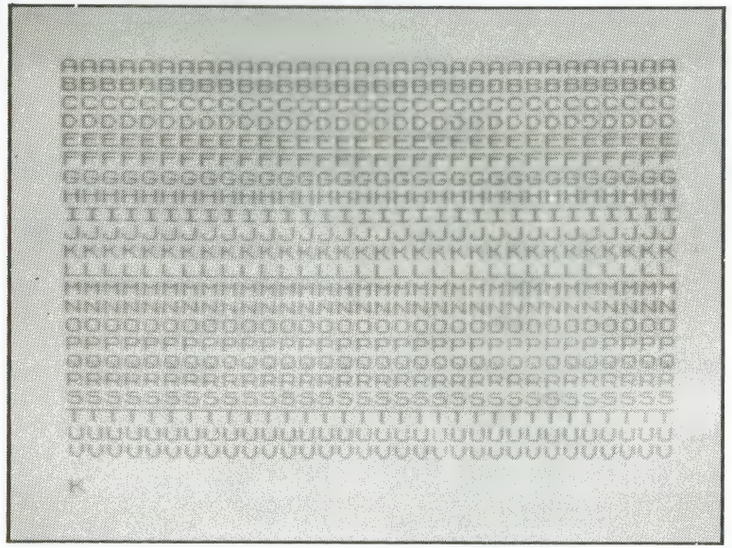


Figure 15.5 Before scrolling a section of screen . . .

Try looping in BASIC: it's very effective. You could use a scroll routine like this, on Block 2, to produce a very nice Fruit Machine program, for example . . .

One final remark. When I first tried this routine, I got the last few lines slightly wrong: I put the POP AF after the JRZ *skip*. The result was that

1. The screen scrolled perfectly,

but

2. I got the error message C: Nonsense in BASIC Ø: 1.

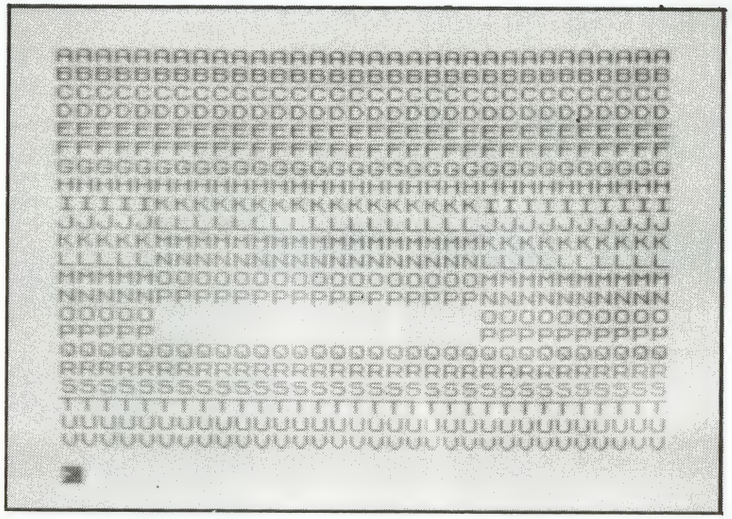


Figure 15.6 . . . and the result, two scrolls later.

This could happen to you too. It's a sign that something has gone wrong with the *stack*. And, in fact, with the code in that order, the final JR *skip* jumps over the POP AF command. So, when the Spectrum tries to return to BASIC, the return address on the stack (which is a 2-byte address) has this residual bit of the AF-register still sitting in it, which confuses the system no end. It's rather too easy to exit from a Machine Code routine with bits of the stack still unPOPped; but it's not a very good idea.

In Machine Code, there are no FOR/NEXT loop commands or IF/THEN conditions. To obtain these, it is necessary to use the flags. As an example, here's a line-renumbering program.

16 More about Flags

I have tried, up till now, to steer clear of the technicalities of the F-register; but every time we've used a conditional jump we've implicitly used the flags. So it's worth taking a closer look. I certainly don't want to go into all of the horrendous details: the exact operation of the F-register is one of the most complicated features of the Z80. But it's not a bad idea to say a bit more.

There is room in the F-register for eight bits; but it only uses six of them. They are:

C	Carry flag
Z	Zero flag
S	Sign flag
P/V	Parity/Overflow flag
H	Half-carry flag
N	Subtract flag

They are arranged in the register like this:

S	Z	X	H	X	P/V	N	C

where "X" means "not used".

The Carry flag is affected mainly by add, subtract, rotate, and shift commands.

The Zero flag is affected by almost everything! Roughly, if anything except LD, INC, DEC changes the contents of A, then the Zero flag is set (to 1) if A is zero, and reset (to 0) otherwise. BIT sets the flag if the specified bit is zero. CP sets or resets the flag according to the result of a comparison.

The Sign flag stores the sign bit of the result of whichever operation has most recently been carried out: 1 for negative, 0 for positive.

The P/V flag works one way for arithmetic, and another for logic. In arithmetic, it is set to 1 if there is an *overflow* in 2's complement arithmetic (e.g. if the sum of two positive numbers runs off the end of the accumulator, giving apparently negative results). For a logical operation it is set to 0 if the byte in A has an *even* number of bits equal to 1; and to 1 if the number of bits equal to 1 is *odd*. This odd/even character of the bits is called the *parity* of the byte. For example.

A holds 01101100: even parity, P/V flag 0
A holds 10010001: odd parity. P/V flag 1.

The H and N flags are used only for binary-coded-decimal calculations, and it's unlikely that you'll need those on a Spectrum. They're mostly for driving other output devices. The way that the C and Z flags are affected by various operations is listed in Appendix 5. These are the most useful flags for the beginner.

We've made use of the Zero flag in a number of routines: every time we've used JRZ, JRNZ, or DJNZ, for example. Usually these conditional jumps are preceded by a *compare* command, such as

```
CP B
```

which *sets the flags* as if it were the command SUB A, B; but doesn't actually alter the contents of A. If A and B hold the same byte, then this subtraction would give 0, and the Zero flag is *set* (to 1). If A and B are different, then the Zero flag is reset.

However, we haven't made much use of the Carry flag. It's especially useful as a loop counter when you don't know for certain that you will hit the exact end of the loop: maybe you'll overshoot.

Specifically, suppose you want to know whether the number held in the HL-register is bigger than that in the BC-register. The command

```
SBC HL, BC          ED 42
```

will set the Carry flag if BC is greater than HL; and will reset it if BC is less than or equal to HL. Follow this by a conditional jump

```
JR C somewhere      38 ??
```

and the jump will occur if BC is bigger than HL. Follow it by

```
JR NC somewhere     30 ??
```

and the jump occurs if BC is equal to HL, or smaller. (For deciding the size, all numbers are considered to be positive: no 2's complement arithmetic!)

LINE RENUMBERING

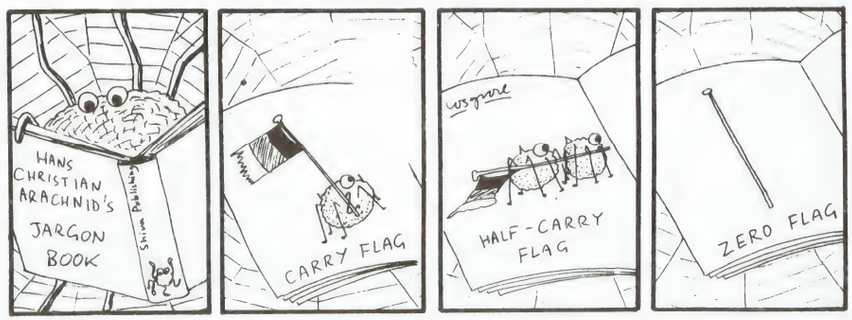
I can illustrate this by a rudimentary line-renumbering routine. All this does is run through the BASIC program area and change the line numbers to the regular sequence 10, 20, 30, . . . going up in 10's until it reaches the end. (A fancier BASIC routine is given in *Further Programming*, and even that could be improved; here I want something easy to understand.)

To do this you need to know how the BASIC lines are stored in memory. I've described all this in *Further Programming*, but to avoid you having to fork out another fiver or more, here it is again.

The BASIC program is stored in memory between the addresses held in PROG and VARS. Without disk drives, PROG is 23755. Each line is held in the form

NS	NJ	LJ	LS	Code for line	ENTER

Here NS and NJ are the senior and junior bytes of the line number; and LJ and LS are the junior and senior bytes of the number of characters in the line, including the ENTER (but not the first four bytes NS, NJ, LJ, LS). Note the senior-first order in the line number: that's *not* a typing error.



For example, the test program

```
1 REM 123456789Ø1
5 PRINT a
```

is held in memory like this:

Address	Contents	Comments
23755	Ø	Senior byte of first line number
23756	1	Junior byte of first line number
23757	13	Junior byte of length of line
23758	Ø	Senior byte of length of line
23759	234	Code for REM
23760	49	Code for 1
23761	50	Code for 2
23762	51	Code for 3
23763	52	Code for 4
23764	53	Code for 5
23765	54	Code for 6
23766	55	Code for 7
23767	56	Code for 8
23768	57	Code for 9
23769	48	Code for Ø
23770	49	Code for 1
23771	13	Code for ENTER
23772	Ø	Senior byte of 2nd line number
23773	5	Junior byte of 2nd line number
23774	3	Junior byte of length of line
23775	Ø	Senior byte of length of line
23776	245	Code for PRINT
23777	97	Code for a
23778	13	Code for ENTER
23779		Start of VARIABLES area

You can check this using the PEEKing routine in *Easy Programming*, page 94. Or write your own. Do it!

Given this, the obvious way to renumber lines is to search through the program area, looking for ENTER bytes (code 13). The next two bytes will be line numbers: change them.

Fine. But it won't work. The reason is that the byte 13 can occur elsewhere; notably in the length-of-line bytes. It does, in address 23757 above (which is why I chose this test program). You *can* get round this, but it's messy. (On a ZX81 the corresponding byte is NEWLINE, code 118. On the whole, lines don't have 118 characters, so it's less of a problem there.)

Further, there's a better way. The length-of-line bytes *tell* you just how far to move to get to the next line-number. Why bother searching for ENTERs?

And that leads to the following code.

THE PROGRAM

The idea is to start at the first two bytes of the program area, first line number; renumber those; use the next two length-of-line bytes to jump ahead to the next line-number; and repeat until you get into the VARS area.

First, we need to store the addresses. We'll use BC to hold the VARS value (where we stop), HL to hold PROG (where we start), and DE to hold the new line number. These are initialized:

```
        LD BC, (VARS)       ED 4B 4B 5C
        LD HL, (PROG)       2A 53 5C
        LD DE, 1Ø dec       11 ØA ØØ
```

For the system variable addresses PROG and VARS see Appendix 3.

Next we want to check to see whether the HL-register (which we'll use as a pointer, running through the program area, because HL is excellent for indirection) exceeds the value in the BC-register. If so, we stop. *That* is where the Carry flag comes in:

```
check:  PUSH HL             E5
        SBC HL, BC          ED 42
        POP HL              E1
        JRNC end            3Ø 15
```

(That last 15 can't actually be worked out until the whole program is written, but that's what it turns out to be.)

Next, the actual renumbering:

```
renumber:  LD (HL), D       72
           INC HL           23
           LD (HL), E       73
```

After that, we have to read the next two bytes to find the length of the line. We have to store the result somewhere. The HL-register is needed for calculations; so the BC-register looks best. Unfortunately it's in use; but we can get round that by stacking the current value for later recovery:

```
        PUSH BC             C5
        INC HL              23
        LD C, (HL)          4E
        INC HL              23
        LD B, (HL)          46
```

Now we shift HL up to the next line-number

```
        ADD HL, BC          Ø9
        INC HL              23
```

and get back the old BC-value

```
        POP BC              C1
```

Finally we want to set DE to the correct value for the next line, by adding 10. That means some more shuffling of registers on to the stack:

PUSH HL	E5
LD HL, 10 dec	21 0A 00
ADD HL, DE	19
LD D, H	54
LD E, L	5D
POP HL	E1

Now loop back to the *check* stage, to repeat unless we've got HL too large:

JR check	18 E5

Lastly, the final RET command appears at the space referred to above as *end*.

```
  5 CLEAR 31999
  6 POKE 23609,50
 10 PRINT "base address ";
 15 DIM h$(2)
 20 INPUT b: PRINT b
 27 PRINT "No. of data bytes ";
 40 INPUT d: PRINT d
 50 FOR i=0 TO d-1
 55 FOR i=0 TO d-1
 60 POKE b+i,0
 70 NEXT i
 80 LET a=b+d
 87 PRINT "CODE "
100 INPUT c$
105 IF c$="s" THEN GO TO 200
110 IF c$="s" THEN GO TO 200
120 PRINT c$+" ";
128 LET hs=CODE c$(1)-48-39*(c$
(1)>"9")
140 LET hj=CODE c$(2)-48-39*(c$
(2)>"9")
150 POKE a,16*hs+hj
BEFORE          L
```

Figure 16.1 Before the line-renumber . . .

```
 10 CLEAR 31999
 20 POKE 23609,50
 30 PRINT "base address ";
 40 DIM h$(2)
 50 INPUT b: PRINT b
 60 PRINT "No. of data bytes ";
 70 INPUT d: PRINT d
 80 FOR i=0 TO d-1
 90 FOR i=0 TO d-1
100 POKE b+i,0
110 NEXT i
120 LET a=b+d
130 PRINT "CODE "
140 INPUT c$
150 IF c$="s" THEN GO TO 200
160 IF c$="s" THEN GO TO 200
170 PRINT c$+" ";
180 LET hs=CODE c$(1)-48-39*(c$
(1)>"9")
190 LET hj=CODE c$(2)-48-39*(c$
(2)>"9")
200 POKE a,16*hs+hj
AFTER           L
```

Figure 16.2 . . . and after. Note that the GO TOs haven't changed.

Load in the entire routine, in the order given (with its final RET as usual). Whatever program you have in the BASIC area, it will now renumber if you enter

RANDOMIZE USR 32000

Of course only the line numbers change: GO TOs and GO SUBs won't match up any more. But the principle is clear enough.

There are some very powerful commands which handle entire blocks of memory at one go:

17 Block Search and Block Transfer

Some of the routines I've shown you, above, are not written in the most efficient way possible. The idea was to keep things as simple as possible to begin with, and I'm not going to apologize for that. Machine Code isn't the easiest thing to tackle, and listing all its features at once is just confusing. On the other hand, if you've looked at other books on Z80 Machine Code, or listings in magazines, you may well be wondering why I've sometimes done things in an apparently cack-handed way. This chapter, and the next, may help redress the balance.

BLOCK SEARCH

First, there are some very powerful instructions which will search a whole block of memory. I'll take CPDR which is short for "compare, decrement and repeat" as an example.

If I write a piece of code like this:

	LD BC, 0100	01 00 01
	LD HL, 5000	21 00 50
	LD A, 05	3E 05
	CPDR	ED B9
next:	——	

what happens is this:

When the CPDR instruction is encountered the value in the A-register is compared with the contents of the byte HL is pointing at. If they are equal, control passes to *next*. If not, BC and HL are both decremented by 1, and the "compare" is repeated until a match is found or until BC contains zero. In other words, those 4 instructions say: "Find the first occurrence of a byte containing 05 from address 5000 (hex) down to address 4F00 and leave HL pointing to it. If there isn't one, zero BC."

So the first example I gave of using jumps, which was a little "compare" loop, could have been done much more easily. But, of course, it wouldn't have illustrated jumps!

There is a companion command CPIR which increments the HL-register instead, and otherwise works the same way. So this searches a block of memory from the other end.

BLOCK TRANSFER

The *block transfer* commands LDIR and LDDR shift blocks of data around in memory. For instance, to use LDIR, you:

1. Load HL with the address of the first byte to be transferred.
2. Load DE with the address of the first destination byte.
3. Load BC with the number of bytes to be moved.

Then LDIR transfers the first byte; increments HL and DE; decrements BC, and keeps doing this until BC hits Ø.

One important point to note is that the final step involves incrementing HL and DE, but not doing the transfer. So at the end, HL is pointing to the byte immediately after the transfer region, and DE is pointing to the top end of the transfer region. See the sideways scrolling routines opposite.

LDDR is similar, but it decrements HL and DE (and continues to decrement BC as for LDIR—BC is just a counter).

PRESET ATTRIBUTES

One way to use block transfer is to set up a "fake" attributes file in RAM, and transfer it into the true attributes file—thereby instantly changing all the attributes to a new, preset pattern. To do this, you need 7Ø4 data bytes to hold the new attributes. This requires a modification to the LOADER program. Edit line 1Ø so that it reads

```
1Ø  CLEAR 31599
```

which leaves plenty of room. Now RUN, and tell it there are 7Ø4 data bytes. The code is:

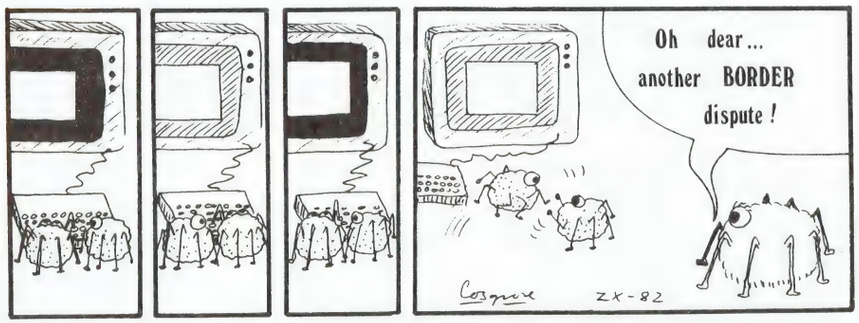
```
LD HL, 316ØØ dec      21 7Ø 7B
LD DE, a-file         11 ØØ 58
LD BC, 7Ø4 dec        Ø1 CØ Ø2
LDIR                  ED BØ
```

The 316ØØ (7B7Ø hex) is the start of the new data area. The routine itself gets loaded into the start address 323Ø4.

Now you need to set up the "fake" file. For example:

```
5ØØØ  FOR w = 316ØØ TO 323Ø3
5Ø1Ø  IF w < 319ØØ THEN POKE w, 48
5Ø2Ø  IF w > = 319ØØ THEN POKE w, 32
5Ø3Ø  NEXT w
```

Press GO TO 5ØØØ (and wait!) to set this up; then get some display on the screen (LIST is an easy way) and finally RANDOMIZE USR 323Ø4. You'll get yellow and green areas of paper, according to the attributes POKEd in by the program at 5ØØØ. No doubt you can think of more exciting BASIC routines to set up the fake file (Stripes? Chequered patterns?).



WHERE WE CAME IN . . .

At this point you should be able to work out what the introductory programs in Chapter 1 were doing—and how. The Machine Code commands in those programs are included in the DATA statements; and they were written in decimal, not hex, for ease of entry. If you translate the decimal numbers into hex and then look them up (the table of opcodes in the *Manual*, page 183, lists them in numerical order) you'll see that they all involve block transfers into the attributes or display files. The bytes being transferred are taken from ROM, so most of the time look pretty random. But some sections of ROM (which?) have rather more structure, and that's where the patterned displays come from.

SIDESCROLL OF ATTRIBUTES

Edit LOADER back to 31999 in line 1Ø (see opposite): we won't need that much memory now.

Presetting the attributes is a very simple use of LDIR. Here's a slightly fancier one: scrolling a row of attributes one space left, and putting the leftmost one at the right hand end, as if the screen were wrapped round like a cylinder. For simplicity, I'll just do row Ø.

```
        LD HL, a-file      21 ØØ 58
loop:   LD D, H            54
        LD E, L            5D
        INC HL             23
        LD BC, 31 dec      Ø1 1F ØØ
        LD A, (DE)         1A
        LDIR               ED BØ
        LD (DE), A         12
```

Set up something to test it on:

```
6ØØØ  FOR s = Ø TO 31: PAPER s/6: PRINT AT Ø, s; "□"; : NEXT s
```

and then RANDOMIZE USR 32ØØØ: watch it scroll. For real action, build a BASIC loop

```
FOR t = 1 TO 5ØØ: RANDOMIZE USR 32ØØØ: NEXT t
```

and watch it whizz round!

By looping this 22 times with suitable starting conditions, you can sidescroll a whole screenful of attributes. I'll tackle a very similar problem: sidescrolling a line of the *display* file. That already involves looping.

SIDESCROLL OF DISPLAY

Here's the code:

```
        LD A, 8 dec        3E Ø8      [loop count]
        LD DE, start       11 ØØ 4Ø   [line start]
set:    LD H, D            62
        LD L, E            6B
        INC HL             23
        PUSH DE            D5         [save line start]
```

```
        LD BC, 31 dec       Ø1 1F ØØ
        PUSH AF             F5          [save counter]
        LD A, (DE)          1A          [save first byte]
        LDIR                ED BØ       [scroll left]
        LD (DE), A          12          [reload first byte]
        POP AF              F1          [recover loop count]
        DEC A               3D          [advance count]
        CP B                B8          [test for end]
        JRZ skip            28 Ø4
        POP DE              D1
        INC D               14          [next line]
        JR set              18 EB
skip:   POP DE              D1
```

Embed that in yet another loop (or loops, block-by-block as well as within an 8-row block) and you'll have a routine to scroll the entire display sideways. Obviously many variants are possible. Study it carefully, and see what changes could be tried.

That's covered most of the Z80 commands other than those that handle peripheral devices. But there's still room to tell you about . . .

18 Some Things I haven't told You about

First things first—it was Humpty-Dumpty.*

BIT-FLIPPING

There are a few simple commands to alter bits that you may well find useful. The first is

CCF 3F

which changes the Carry flag from Ø to 1 or vice versa. To set the Carry flag to 1, use

SCF 37

To *reset* the Carry flag (to Ø) use both of these in turn: SCF, then CCF.

To complement the entire A-register (i.e. flip every bit to its opposite) use

CPL 2F

MNEMONICS

The next thing I should mention is that some people use slightly different mnemonic opcodes from those I've described. For instance, where I would write LD A, (nn) some people write LD (nn). This is because the A-register is the only register which can be loaded directly, so it's not strictly necessary to specify it. I find that actually quoting it every time is a useful aid to memory, though.

ALTERNATE REGISTERS

I said that there is an alternate set of registers, and then promptly ignored them. You can always get away without using them, and they aren't very useful anyway. You can't do arithmetic in them. Their main use is to save temporarily the contents of the main set while you're executing some routine which alters the main register contents in ways you

* See page 17.

don't want. This is done by exchanging the contents of the main and alternate sets before, and again after, the offending routine:

```
EX AF, AF'      08      swap AF with AF'
EXX             D9      swap BC, DE, HL with BC', DE', HL'
CALL . . .      CD——    call offending routine
EX AF, AF'      08  ┐
EXX             D9  ┘—  restore registers
```

Of course, you could do the same thing by PUSHing the register contents you want saved on to the stack before the CALL, and POPping them off afterwards.

Never use the IY register. The Spectrum needs it!

ROM ROUTINES

I have also been guilty of reinventing the wheel now and then. The point is that the BASIC interpreter in ROM has to call on routines such as those we've developed. So why not simply *call* them, rather than write our own? In general, the answer is that it would have been much more sensible to do so, since it saves a lot of effort and, almost as important, computer memory. But—so far as *this* book is concerned, my aim has been to tell you about Z80 machine code, and avoid, as far as possible, the special features of the Spectrum. If all the examples had consisted of a series of calls to addresses in ROM you wouldn't have learnt much!

SAVING BASIC + MACHINE CODE

Suppose you've got a BASIC program that makes use of a Machine Code routine stored above RAMTOP (say at 32000). How can you usefully SAVE both?

One way is to use, in turn:

```
SAVE "program"
SAVE "m-code" CODE 32000, 600
```

(where 600 is the length of the Machine Code area: the actual length of the code is more efficient but that's up to you). Then you have to LOAD in turn using

```
LOAD "program"
```

and when that's in

```
LOAD "m-code" CODE
```

Don't forget to CLEAR 32000 too.

Better still is to write a little BASIC routine in the program to do all this for you. Find a spare line—say at the end—and write

```
9800  LOAD "m-code" CODE
```

or whichever is the start line of the BASIC program. Now do


```
SAVE "program" LINE 9800
SAVE "m-code" CODE 32000, 600
```

manually. Then if you, as usual, type

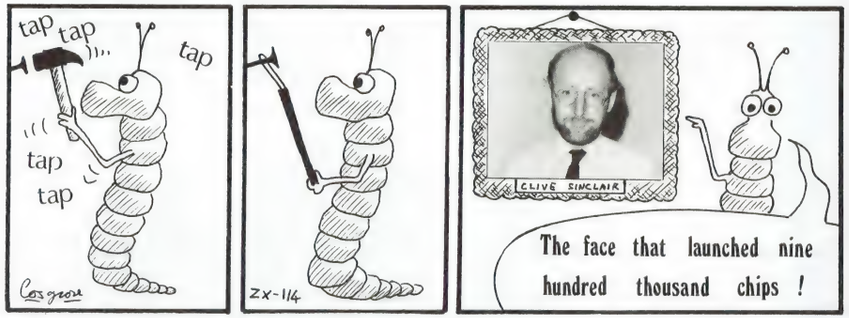
```
LOAD "program"
```

it loads, starts running at line 9800, loads in the Machine Code, and continues.

You can even write the SAVE as a BASIC routine too, if you want. Experiment.

MULTIPLE ROUTINES

You can string several different Machine Code routines end-to-end, provided you put RET commands at the end of each; and each can be called by RANDOMIZE USR (start address of routine required). Note that you can save the entire block of such Machine Code in one go: it's *not* necessary to save routine by routine.



EFFICIENT USE OF MACHINE CODE

I want to tie up two loose ends I left hanging right at the beginning. I said that there may be reasons other than BASIC's need to interpret statements each time they are executed for a Machine Code program to run faster than BASIC. I'll explain with an example:

```
BASIC                                   Machine Code
10  FOR i = 20 TO 1 STEP -1             LD B, 14
    .....                       loop:   .....
50  NEXT i                              DJNZ loop
```

In each case, every time the loop is executed a variable is decremented by 1. But that process is much more complicated in BASIC than it is in Machine Code. The reason is that since BASIC has to deal with decimal values some of the time, it assumes that it's doing so all the time, and so it actually subtracts 1.00000000, which is no easier than subtracting 1.58712684. In fact, the procedure employed is quite complex and time-consuming. The Machine Code, on the other hand, uses a single, purpose-built instruction. The result is in the region of 100 times faster.

The other point I deferred was that Machine Code can occupy more memory than its BASIC equivalent. Here's an example which illustrates why:

BASIC		Machine Code	No. of bytes	
30	IF r = p AND p = q THEN			
	LET p = w	LD HL, 50000	3	
		LD A, (HL)	1	
		LD HL, 50001	3	
		SUB A, (HL)	1	
		JRNZ nextbit	2	
		LD HL, 50001	3	
		LD A, (HL)	1	
		LD HL, 50002	3	
		SUB A, (HL)	1	
		JRNZ nextbit	2	
		LD HL, 50001	3	
		LD A, (50003)	3	
		LD (HL), A	1	
	nextbit:		27	TOTAL

The Machine Code assumes that r, p, q and w are held in the bytes 50000, 50001, 50002 and 50003, respectively. In practice it wouldn't be as simple as that because each number will occupy 5 bytes and the SUB will actually be a CALL to a floating point subtraction routine. In any event, the actual code would need at least as many, and probably more than, the 27 bytes shown. The equivalent BASIC line occupies only 18 bytes, 1 for each of the symbols (IF, =, w, AND, and so on), 4 for the line number and 1 for the line delimiter. The more complex the BASIC statement, the more memory overhead there is in the Machine Code version.

OTHER PLACES TO STORE MACHINE CODE

The main disadvantage with storing code above RAMTOP is that you can't save it directly as part of a BASIC program. The advantage is that you *can* load other stuff in under it. But, there *are* alternatives.

A favourite trick is to store the whole thing in a REM statement, the first line in the BASIC program. The first character after the REM normally has address 23755. So you write your BASIC starting with

```
1 REM XXXXXX . . . X
```

with enough Xs to hold the code; and then POKE the Machine Code in. The code is accessed by RANDOMIZE USR 23755 (or LET y = USR 23755, etc.) and can be saved; also it is not destroyed by RUN.

Another place to store the code is in a character string, which is easily located (via the system variable VARS) provided you make this the first variable declared—e.g. start with a big enough string array:

```
1 DIM a$ (79)          [to hold 79 bytes]
```

and then put the machine code into a$ (1), a$ (2), etc. as you build up the string. The main disadvantage is that RUN or CLEAR will wipe out your code. And the start address can sometimes wander. You can also store it as DATA and load above RAMTOP as part of the BASIC, as in Chapter 1, but that way wastes a lot of space.

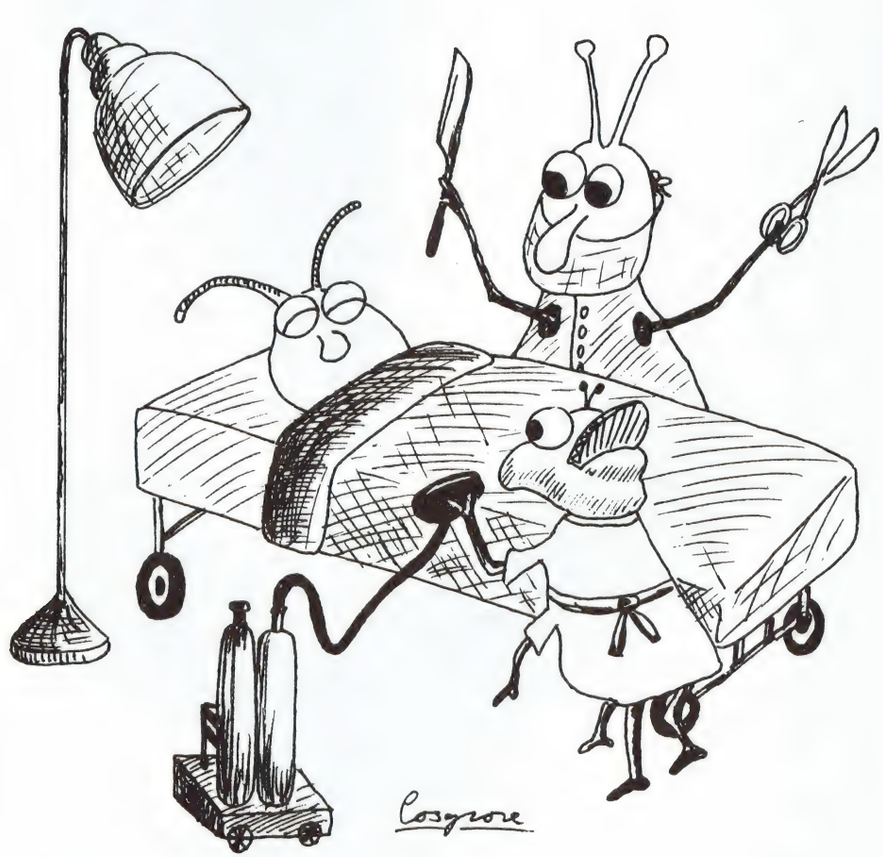
DEBUGGING

There are no built-in debugging facilities in Machine Code; and while HELPA will help you edit code, it isn't really designed to debug. (Nor are several other programs advertised as useful for "debugging" Machine Code!) Your best bet at this stage is to *dry-run* the routine using old-fashioned pencil and paper (see *Easy Programming*, page 39). Of course you can insert tracing statements into Machine Code, but watch out for changes in addresses and jump-sizes.

One useful (though apparently cack-handed) trick is to write your routine in BASIC first, and debug *that*: use only BASIC instructions that correspond to the Machine Code. (That is, *emulate* the Machine Code in BASIC.) It will work slowly, if at all; but it's debuggable.

For the really ambitious, this suggests a project. The idea is to improve HELPA by adding a routine SINGLE-STEP which will work through the program command by command, displaying the registers on the screen. You'll need to (*a*) write a Machine Code routine to PUSH all registers on to the stack, then display them on the screen in hex; (*b*) add to this a routine that looks for a keyboard input; (*c*) write between each line of machine code a CALL to this routine (use the ** delimiters in HELPA to signal where); (*d*) work out how to return to BASIC if you want.

Appendices



1 Hex/Decimal Conversion

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
8	−128	−127	−126	−125	−124	−123	−122	−121	−120	−119	−118	−117	−116	−115	−114	−113
9	−112	−111	−110	−109	−108	−107	−106	−105	−104	−103	−102	−101	−100	−99	−98	−97
A	−96	−95	−94	−93	−92	−91	−90	−89	−88	−87	−86	−85	−84	−83	−82	−81
B	−80	−79	−78	−77	−76	−75	−74	−73	−72	−71	−70	−69	−68	−67	−66	−65
C	−64	−63	−62	−61	−60	−59	−58	−57	−56	−55	−54	−53	−52	−51	−50	−49
D	−48	−47	−46	−45	−44	−43	−42	−41	−40	−39	−38	−37	−36	−35	−34	−33
E	−32	−31	−30	−29	−28	−27	−26	−25	−24	−23	−22	−21	−20	−19	−18	−17
F	−16	−15	−14	−13	−12	−11	−10	−9	−8	−7	−6	−5	−4	−3	−2	−1
0	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
2	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
3	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
4	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
5	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95
6	96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111
7	112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127
8	128	129	130	131	132	133	134	135	136	137	138	139	140	141	142	143
9	144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159
A	160	161	162	163	164	165	166	167	168	169	170	171	172	173	174	175
B	176	177	178	179	180	181	182	183	184	185	186	187	188	189	190	191
C	192	193	194	195	196	197	198	199	200	201	202	203	204	205	206	207
D	208	209	210	211	212	213	214	215	216	217	218	219	220	221	222	223
E	224	225	226	227	228	229	230	231	232	233	234	235	236	237	238	239
F	240	241	242	243	244	245	246	247	248	249	250	251	252	253	254	255

2's complement

ordinary

2 Memory Reservation Tables

To reserve memory in 100-byte blocks at the top end of RAM, use the following addresses. Remember to CLEAR from the address one below that which is actually required.

Number of bytes reserved	16K		48K	
	Decimal address	Hex address	Decimal address	Hex address
100	32500	7EF4	65268	FEF4
200	32400	7E90	65168	FE90
300	32300	7E2C	65068	FE2C
400	32200	7DC8	64968	FDC8
500	32100	7D64	64868	FD64
600	32000	7D00	64768	FD00
700	31900	7C9C	64668	FC9C
800	31800	7C38	64568	FC38
900	31700	7BD4	64468	FBD4
1000	31600	7B70	64368	FB70
1100	31500	7B0C	64268	FB0C
1200	31400	7AA8	64168	FAA8

3 System Variable Addresses

POINTERS TO MOVEABLE ADDRESSES

	Decimal	Hex
CHANS	23631	5C4F
PROG	23635	5C53
VARS	23627	5C4B
E-LINE	23641	5C59
WORKSP	23649	5C61
STKBOT	23651	5C63
STKEND	23653	5C65
RAMTOP	23730	5CB2
UDG	23675	5C7B
P-RAMT	23732	5CB4
CHARS	23606	5C36
BORDCR	23624	5C48
DFSZ	23659	5C6B
COORDS	23677–8	5C7D–E
DFCC	23684	5C84
DFCC-S	23685	5C85
SCR-CT	23692	5C8C
ATTR-P	23693	5C8D

FIXED ADDRESSES

	Decimal	Hex
DISPLAY FILE	16384	4000
ATTRIBUTES FILE	22528	5800
16K RAMTOP VALUE	32599	7F57
48K RAMTOP VALUE	65367	FF57
16K UDG VALUE	32600	7F58
48K UDG VALUE	65368	FF58
16K P-RAMT VALUE	32767	7FFF
48K P-RAMT VALUE	65535	FFFF
PRINTER BUFFER	23296	5B00
SYSTEM VARIABLES	23552	5C00
MICRODRIVE MAPS	23734	5CB6
CHARACTER SET	15360	3C00

4 Summary of Z80 Commands

This is a list of all the opcode mnemonics, with a summary of their effects. Those commands explained more fully in the text are given a page reference. The effects on the flags are omitted: for these, consult the Zilog Reference Card or the books listed in the bibliography under "Machine Code".

ADC	Page 43	Add, including the carry flag. Store in A or HL.
ADD	Page 36	Add, ignoring the carry flag. Store in A or HL.
AND	Page 40	Logic AND on corresponding bits: store in A.
BIT	Page 50	BIT b, r sets the Zero flag according to the value of the b-th bit of the byte in register r. The bits are in the order 76543210 within each byte.
CALL	Page 20	Calls a subroutine. There are conditional calls signalled by the additional letters C (call if the Carry flag is set); M (if the Sign flag is set—"the result (of a compare) is negative"); NC (if the Carry flag is not set); NZ (if the Zero flag is not set); P (if the Sign flag is not set—"the result is positive"); PE (if the Parity flag is set: ignore this one); PO (if the Parity flag is not set: ignore this too); Z(if the Zero flag is set). For flags, see Page 70, 92.
CCF	Page 79	Complement Carry flag (i.e. swap 0 and 1).
CP	Page 40	Compare: sets the flags as if it were a subtraction from A, but leaves A unchanged.
CPD		Compare and decrement. Compare through HL; then decrement HL and BC.
CPDR	Page 75	Compare, decrement, repeat: block search. Like CPD but repeating until either the result of the comparison is 0, or BC reaches 0.
CPI		Like CPD except that HL increments; BC still decrements.
CPIR	Page 75	Like CPDR but incrementing HL.
CPL	Page 79	Complement (flip bits of) the A-register.
DAA		Decimal adjust accumulator. Used in binary-coded decimal work: ignore.
DEC	Page 42	Decrement: reduce value by 1.
DI		Disable interrupts. Ignore.
DJNZ	Page 43	Decrement, jump if non-zero. Decrement B and jump relative unless the Zero flag is set. Used in loops like a BASIC FOR/NEXT.
EI		Enable interrupts. Ignore.
EX	Page 80	Exchange values. Instructions with (SP) exchange the registers HL, IX or IY with the top of the stack.
EXX	Page 80	Exchange all three register-pairs BC, DE, HL with their alternates BC', DE', HL'.
HALT		Wait for an interrupt. Unless you've got hardware attached, and know what you're doing, DO NOT USE as the program will wait forever.
IM		Interrupt mode: ignore.
IN		Input from a device. Ignore.
INC	Page 42	Increment: increase value by 1.

IND, INDR, INI, INIR		Input commands analogous to LDD, LDDR, LDI, LDIR. Ignore.
JP	Page 41	Jump. Variants with added C, M, NC, NZ, P, PE, PO, Z are conditional jumps, with conditions as for CALL.
JR	Page 41	Jump relative—followed by a 1-byte displacement. Conditional variants are C, NC, NZ, Z only.
LD	Page 29	Load. Can use all five addressing modes.
LDD		Not the same as LD D! Load what HL points to into what DE points to: decrement BC, DE, HL.
LDDR	Page 75	Load, decrement, repeat: block transfer. Do LDD until BC hits zero. Copies a block of memory whose length is stored in BC, out of what HL points to and into what DE points to.
LDI		Like LDD except that HL and DE increment: BC still decrements.
LDIR	Page 75	Like LDDR except that HL and DE increment.
NEG		Negative: change the sign of the contents of A.
NOP		No operation. Do nothing for 1 time-cycle—i.e. waste time. Useful for temporary deletion of instructions when debugging: harmless and helpful.
OR	Page 40	Logic OR on bits. Store in A.
OTDR, OTIR, OUT, OUTD, OUTI		Various outputs. Ignore.
POP	Page 44	Pop from stack into indicated register.
PUSH	Page 44	Push from register on to stack.
RES	Page 50	Reset a bit—i.e. make it zero.
RET	Page 22	Return from subroutine. Conditional returns, corresponding to the possibilities for CALL, are possible. (Conditions on a CALL need not match those on a RET!)
RETI, RETN		Return from interrupt subroutines. Ignore.
RL		Rotate left: like a shift, except that the carry flag is included as if it were bit number 8.
RLA		Rotate left accumulator. Like RL A but with a different effect on the flags.
RLC		Not the same as RL C! Rotate left, but put bit 7 into carry *and* into bit 0.
RLCA		Like RLC A, but same flag difference as RLA.
RLD		Not what you'd expect at all: rotate left decimal. Used for binary coded decimal: ignore.
RR		Like RL but to the right.
RRA		Like RLA.
RRC		Like RLC.
RRCA		Like RLCA.
RRD		Like RLD.
RST		Like CALL, but only from addresses 0, 8, 10, 18, 20, 28, 30, 38 (hex). These are all in the ROM on the Spectrum: see Ian Logan's books in the Bibliography. RST 0 is like temporarily disconnecting the power.
SBC	Page 43	Subtract, taking account of the carry flag. Store in A or HL.
SCF	Page 79	Set carry flag (to 1).
SET	Page 58	Set a bit—i.e. make it 1.
SLA	Page 43	Shift left arithmetic. All bits move up 1; bit 0 becomes 0.
SRA	Page 43	Shift right arithmetic. Move bits down 1; copy bit 7 into 6 *and* 7.
SRL	Page 43	Shift right logical. Move bits down 1 place; make bit 7 zero.
SUB	Page 36	Subtract, ignoring carry. Store in A. (There is no SUB HL, r command: if you want one, reset the Carry flag and use SBC.
XOR	Page 40	Exclusive or on each bit. Store in A.

5 Zero and Carry Flags

This appendix shows which instructions affect the Zero and Carry flags. The symbols have the following meanings:

–	unchanged
*	affected by the result of the operation
1	set to 1
Ø	reset to Ø
+	set to 1 if A = (HL), reset to Ø otherwise

Instruction	C	Z	Instruction	C	Z	Instruction	C	Z
ADC	*	*	IND	–	*	RET	–	–
ADD (8-bit)	*	*	INDR	–	1	RETI	–	–
ADD (16-bit)	*	–	INI	–	*	RETN	–	–
AND	Ø	*	INIR	–	1	RL	*	*
BIT	–	*	JP	–	–	RLA	*	–
CALL	*	*	JR	–	–	RLC	*	*
CCF	*	–	LD A, I	–	*	RLCA	*	–
CP	*	*	LD A, R	–	*	RLD	–	*
CPD	–	+	LD (all others)	–	–	RR	*	*
CPDR	–	+	LDD	–	–	RRA	*	–
CPI	–	+	LDDR	–	–	RRC	*	*
CPIR	–	+	LDI	–	–	RRCA	*	–
CPL	–	–	LDIR	–	–	RRD	–	*
DAA	*	*	NEG	*	*	RST	–	–
DEC	–	–	NOP	–	–	SBC	*	*
DI	–	–	OR	Ø	*	SCF	1	–
DJNZ	–	–	OTDR	–	1	SET	–	–
EI	–	–	OTIR	–	1	SLA	*	*
EX	–	–	OUT	–	–	SRA	*	*
EXX	–	–	OUTD	–	*	SRL	*	*
HALT	–	–	OUTI	–	*	SUB	*	*
IM	–	–	POP	–	–	XOR	Ø	*
IN A (n)	–	–	PUSH	–	–			
IN r, (C)	¬	*	RES	–	–			

6 Z80 Opcodes

This is a complete list of Z80 opcodes, arranged alphabetically by mnemonic. In the listing, the symbol n stands for any single-byte number; nn for any 2-byte number; and d is 1-byte displacement written in 2's complement notation. Note that all 2-byte numbers are encoded with the junior byte *preceding* the senior.

Examples:

LD BC, nn has the opcode Ø1 nn, so LD BC, 732F codes as Ø1 2F 73.

LD A, (IY + d) " " " FD 7E d, so LD A (IY + Ø7) " " FD 7E Ø7.

The table of opcodes is based on one published by Zilog Inc. A listing numerically by opcode is included in the Sinclair *Manual* Appendix A: note the use of lower case letters there for mnemonics.

ADC A, (HL)	8E	BIT 1, A	CB4F	BIT 7, C	CB79
ADC A, (IX + d)	DD8Ed	BIT 1, B	CB48	BIT 7, D	CB7A
ADC A, (IY + d)	FD8Ed	BIT 1, C	CB49	BIT 7, E	CB7B
ADC A, A	8F	BIT 1, D	CB4A	BIT 7, H	CB7C
ADC A, B	88	BIT 1, E	CB4B	BIT 7, L	CB7D
ADC A, C	89	BIT 1, H	CB4C	CALL C, nn	DCnn
ADC A, D	8A	BIT 1, L	CB4D	CALL M, nn	FCnn
ADC A, E	8B	BIT 2, (HL)	CB56	CALL NC, nn	D4nn
ADC A, H	8C	BIT 2, (IX + d)	DDCBd56	CALL nn	CDnn
ADC A, L	8D	BIT 2, (IY + d)	FDCBd56	CALL NZ, nn	C4nn
ADC A, n	CEn	BIT 2, A	CB57	CALL P, nn	F4nn
ADC HL, BC	ED4A	BIT 2, B	CB50	CALL PE, nn	ECnn
ADC HL, DE	ED5A	BIT 2, C	CB51	CALL PO, nn	E4nn
ADC HL, HL	ED6A	BIT 2, D	CB52	CALL Z, nn	CCnn
ADC HL, SP	ED7A	BIT 2, E	CB53	CCF	3F
ADD A, (HL)	86	BIT 2, H	CB54	CP (HL)	BE
ADD A, (IX + d)	DD86d	BIT 2, L	CB55	CP (IX + d)	DDBEd
ADD A, (IY + d)	FD86d	BIT 3, (HL)	CB5E	CP (IY + d)	FDBEd
ADD A, A	87	BIT 3, (IX + d)	DDCBd5E	CPA	BF
ADD A, B	80	BIT 3, (IY + d)	FDCBd5E	CP B	B8
ADD A, C	81	BIT 3, A	CB5F	CP C	B9
ADD A,D	82	BIT 3, B	CB58	CP D	BA
ADD A, E	83	BIT 3, C	CB59	CP E	BB
ADD A, H	84	BIT 3, D	CB5A	CP H	BC
ADD A, L	85	BIT 3, E	CB5B	CP L	BD
ADD A, n	C6n	BIT 3, H	CB5C	CP n	FEn
ADD HL, BC	09	BIT 3, L	CB5D	CPD	EDA9
ADD HL, DE	19	BIT 4, (HL)	CB66	CPDR	EDB9
ADD HL, HL	29	BIT 4, (IX + d)	DDCBd66	CPI	EDA1
ADD HL, SP	39	BIT 4, (IY + d)	FDCBd66	CPIR	EDB1
ADD IX, BC	DD09	BIT 4, A	CB67	CPL	2F
ADD IX, DE	DD19	BIT 4, B	CB60	DAA	27
ADD IX, IX	DD29	BIT 4, C	CB61	DEC (HL)	35
ADD IX, SP	DD39	BIT 4, D	CB62	DEC (IX + d)	DD35d
ADD IY, BC	FD09	BIT 4, E	CB63	DEC (IY + d)	FD35d
ADD IY, DE	FD19	BIT 4, H	CB64	DEC A	3D
ADD IY, IY	FD29	BIT 4, L	CB65	DEC B	05
ADD IY, SP	FD39	BIT 5, (HL)	CB6E	DEC BC	0B
AND (HL)	A6	BIT 5, (IX + d)	DDCBd6E	DEC C	0D
AND (IX + d)	DDA6d	BIT 5, (IY + d)	FDCBd6E	DEC D	15
AND (IY + d)	FDA6d	BIT 5, A	CB6F	DEC DE	1B
AND A	A7	BIT 5, B	CB68	DEC E	1D
AND B	A0	BIT 5, C	CB69	DEC H	25
AND C	A1	BIT 5, D	CB6A	DEC HL	2B
AND D	A2	BIT 5, E	CB6B	DEC IX	DD2B
AND E	A3	BIT 5, H	CB6C	DEC IY	FD2B
AND H	A4	BIT 5, L	CB6D	DEC L	2D
AND L	A5	BIT 6, (HL)	CB76	DEC SP	3B
AND n	E6n	BIT 6, (IX + d)	DDCBd76	DI	F3
BIT 0, (HL)	CB46	BIT 6, (IY + d)	FDCBd76	DJNZ, d	10d
BIT 0, (IX + d)	DDCBd46	BIT 6, A	CB77	EI	FB
BIT 0, (IY + d)	FDCBd46	BIT 6, B	CB70	EX (SP), HL	E3
BIT 0, A	CB47	BIT 6, C	CB71	EX (SP), IX	DDE3
BIT 0, B	CB40	BIT 6, D	CB72	EX (SP), IY	FDE3
BIT 0, C	CB41	BIT 6, E	CB73	EX AF, AF'	08
BIT 0, D	CB42	BIT 6, H	CB74	EX DE, HL	EB
BIT 0, E	CB43	BIT 6, L	CB75	EXX	D9
BIT 0, H	CB44	BIT 7, (HL)	CB7E	HALT	76
BIT 0, L	CB45	BIT 7, (IX + d)	DDCBd7E	IM 0	ED46
BIT 1, (HL)	CB4E	BIT 7, (IY + d)	FDCBd7E	IM 1	ED56
BIT 1, (IX + d)	DDCBd4E	BIT 7, A	CB7F	IM 2	ED5E
BIT 1, (IY + d)	FDCBd4E	BIT 7, B	CB78	IN A, (C)	ED78

IN A, (n)	DBn
IN B, (C)	ED40
IN C, (C)	ED48
IN D, (C)	ED50
IN E, (C)	ED58
IN H, (C)	ED60
IN L, (C)	ED68
INC (HL)	34
INC (IX + d)	DD34d
INC (IY + d)	FD34d
INC A	3C
INC B	04
INC BC	03
INC C	0C
INC D	14
INC DE	13
INC E	1C
INC H	24
INC HL	23
INC IX	DD23
INC IY	FD23
INC L	2C
INC SP	33
IND	EDAA
INDR	EDBA
INI	EDA2
INIR	EDB2
JP (HL)	E9
JP (IX)	DDE9
JP (IY)	FDE9
JP C, nn	DAnn
JP M, nn	FAnn
JP NC, nn	D2nn
JP nn	C3nn
JP NZ, nn	C2nn
JP P, nn	F2nn
JP PE, nn	EAnn
JP PO, nn	E2nn
JP Z, nn	CAnn
JR C, d	38d
JR, d	18d
JR NC, d	30d
JR NZ, d	20d
JR Z, d	28d
LD (BC), A	02
LD (DE), A	12
LD (HL), A	77
LD (HL), B	70
LD (HL), C	71
LD (HL), D	72
LD (HL), E	73
LD (HL), H	74
LD (HL), L	75
LD (HL), n	36n
LD (IX + d), A	DD77d
LD (IX + d), B	DD70d
LD (IX + d), C	DD71d
LD (IX + d), D	DD72d
LD (IX + d), E	DD73d
LD (IX + d), H	DD74d
LD (IX + d), L	DD75d
LD (IX + d), n	DD36dn
LD (IY + d), A	FD77d
LD (IY + d), B	FD70d
LD (IY + d), C	FD71d
LD (IY + d), D	FD72d
LD (IY + d), E	FD73d
LD (IY + d), H	FD74d
LD (IY + d), L	FD75d
LD (IY + d), n	FD36dn
LD (nn), A	32nn
LD (nn), BC	ED43nn
LD (nn), DE	ED53nn
LD (nn), HL	22nn
LD (nn), IX	DD22nn
LD (nn), IY	FD22nn
LD (nn), SP	ED73nn
LD A, (BC)	0A
LD A, (DE)	1A
LD A, (HL)	7E
LD A, (IX + d)	DD7Ed
LD A, (IY + d)	FD7Ed
LD A, (nn)	3Ann
LD A, A	7F
LD A, B	78
LD A, C	79
LD A, D	7A
LD A, E	7B
LD A, H	7C
LD A, I	ED57
LD A, L	7D
LD A, n	3En
LD B, (HL)	46
LD B, (IX + d)	DD46d
LD B, (IY + d)	FD46d
LD B, A	47
LD B, B	40
LD B, C	41
LD B, D	42
LD B, E	43
LD B, H	44
LD B, L	45
LD B, n	06n
LD BC, (nn)	ED4Bnn
LD BC, nn	01nn
LD C, (HL)	4E
LD C, (IX + d)	DD4Ed
LD C, (IY + d)	FD4Ed
LD C, A	4F
LD C, B	48
LD C, C	49
LD C, D	4A
LD C, E	4B
LD C, H	4C
LD C, L	4D
LD C, n	0En
LD D, (HL)	56
LD D, (IX + d)	DD56d
LD D, (IY + d)	FD56d
LD D, A	57
LD D, B	50
LD D, C	51
LD D, D	52
LD D, E	53
LD D, H	54
LD D, L	55
LD D, n	16n
LD DE, (nn)	ED5Bnn
LD DE, nn	11nn
LD E, (HL)	5E
LD E, (IX + d)	DD5Ed
LD E, (IY + d)	FD5Ed
LD E, A	5F
LD E, B	58
LD E, C	59
LD E, D	5A
LD E, E	5B
LD E, H	5C
LD E, L	5D
LD E, n	1En
LD H, (HL)	66
LD H, (IX + d)	DD66d
LD H, (IY + d)	FD66d
LD H, A	67
LD H, B	60
LD H, C	61
LD H, D	62
LD H, E	63
LD H, H	64
LD H, L	65
LD H, n	26n
LD HL, (nn)	2Ann
LD HL, nn	21nn
LD I, A	ED47
LD IX, (nn)	DD2Ann
LD IX, nn	DD21nn
LD IY, (nn)	FD2Ann
LD IY, nn	FD21nn
LD L, (HL)	6E
LD L, (IX + d)	DD6Ed
LD L, (IY + d)	FD6Ed
LD L, A	6F
LD L, B	68
LD L, C	69
LD L, D	6A
LD L, E	6B
LD L, H	6C
LD L, L	6D
LD L, n	2En
LD SP, (nn)	ED7Bnn
LD SP, HL	F9
LD SP, IX	DDF9
LD SP, IY	FDF9
LD SP, nn	31nn
LDD	EDA8
LDDR	EDB8
LDI	EDA0
LDIR	EDB0
NEG	ED44
NOP	00
OR (HL)	B6
OR (IX + d)	DDB6d
OR (IY + d)	FDB6d
OR A	B7
OR B	B0
OR C	B1
OR D	B2
OR E	B3
OR H	B4
OR L	B5
OR n	F6n
OTDR	EDBB
OTIR	EDB3
OUT (C), A	ED79
OUT (C), B	ED41
OUT (C), C	ED49
OUT (C), D	ED51
OUT (C), E	ED59
OUT (C), H	ED61
OUT (C), L	ED69
OUT (n), A	D3n
OUTD	EDAB
OUTI	EDA3
POPAF	F1
POP BC	C1
POP DE	D1
POP HL	E1
POP IX	DDE1
POP IY	FDE1
PUSH AF	F5
PUSH BC	C5
PUSH DE	D5
PUSH HL	E5
PUSH IX	DDE5
PUSH IY	FDE5
RES 0, (HL)	CB86
RES 0, (IX + d)	DDCBd86
RES 0, (IY + d)	FDCBd86
RES 0, A	CB87
RES 0, B	CB80
RES 0, C	CB81
RES 0, D	CB92
RES 0, E	CB83
RES 0, H	CB84
RES 0, L	CB85
RES 1, (HL)	CB8E
RES 1, (IX + d)	DDCBd8E
RES 1, (IY + d)	FDCBd8E
RES 1, A	CB8F
RES 1, B	CB88
RES 1, C	CB89
RES 1, D	CB8A
RES 1, E	CB8B
RES 1, H	CB8C
RES 1, L	CB8D
RES 2, (HL)	CB96
RES 2, (IX + d)	DDCBd96
RES 2, (IY + d)	FDCBd96
RES 2, A	CB97
RES 2, B	CB90
RES 2, C	CB91
RES 2, D	CB92
RES 2, E	CB93
RES 2, H	CB94
RES 2, L	CB95
RES 3, (HL)	CB9E
RES 3, (IX + d)	DDCBd9E
RES 3, (IY + d)	FDCBd9E
RES 3, A	CB9F
RES 3, B	CB98
RES 3, C	CB99
RES 3, D	CB9A
RES 3, E	CB9B
RES 3, H	CB9C
RES 3, L	CB9D

RES 4, (HL)	CBA6
RES 4, (IX + d)	DDCBdA6
RES 4, (IY + d)	FDCBdA6
RES 4, A	CBA7
RES 4, B	CBA0
RES 4, C	CBA1
RES 4, D	CBA2
RES 4, E	CBA3
RES 4, H	CBA4
RES 4, L	CBA5
RES 5, (HL)	CBAE
RES 5, (IX + d)	DDCBdAE
RES 5, (IY + d)	FDCBdAE
RES 5, A	CBAF
RES 5, B	CBA8
RES 5, C	CBA9
RES 5, D	CBAA
RES 5, E	CBAB
RES 5, H	CBAC
RES 5, L	CBAD
RES 6, (HL)	CBB6
RES 6, (IX + d)	DDCBdB6
RES 6, (IY + d)	FDCBdB6
RES 6, A	CBB7
RES 6, B	CBB0
RES 6, C	CBB1
RES 6, D	CBB2
RES 6, E	CBB3
RES 6, H	CBB4
RES 6, L	CBB5
RES 7, (HL)	CBBE
RES 7, (IX + d)	DDCBdBE
RES 7, (IY + d)	FDCBdBE
RES 7, A	CBBF
RES 7, B	CBB8
RES 7, C	CBB9
RES 7, D	CBBA
RES 7, E	CBBB
RES 7, H	CBBC
RES 7, L	CBBD
RET	C9
RET C	D8
RET M	F8
RET NC	D0
RET NZ	C0
RET P	F0
RET PE	E8
RET PO	E0
RET Z	C8
RETI	ED4D
RETN	ED45
RL (HL)	CB16
RL (IX + d)	DDCBd16
RL (IY + d)	FDCBd16
RL A	CB17
RL B	CB10
RL C	CB11
RL D	CB12
RL E	CB13
RL H	CB14
RL L	CB15
RLA	17
RLC (HL)	CB06
RLC (IX + d)	DDCBd06
RLC (IY + d)	FDCBd06
RLC A	CB07
RLC B	CB00
RLC C	CB01
RLC D	CB02
RLC E	CB03
RLC H	CB04
RLC L	CB05
RLCA	07
RLD	ED6F
RR (HL)	CB1E
RR (IX + d)	DDCBd1E
RR (IY + d)	FDCBd1E
RR A	CB1F
RR B	CB18
RR C	CB19
RR D	CB1A
RR E	CB1B
RR H	CB1C
RR L	CB1D
RRA	1F
RRC (HL)	CB0E
RRC (IX + d)	DDCBd0E
RRC (IY + d)	FDCBd0E
RRC A	CB0F
RRC B	CB08
RRC C	CB09
RRC D	CB0A
RRC E	CB0B
RRC H	CB0C
RRC L	CB0D
RRC A	0F
RRD	ED67
RST 0	C7
RST 10H	D7
RST 18H	DF
RST 20H	E7
RST 28H	EF
RST 30H	F7
RST 38H	FF
RST 8	CF
SBC A, (HL)	9E
SBC A, (IX + d)	DD9Ed
SBC A, (IY + d)	FD9Ed
SBC A, A	9F
SBC A, B	98
SBC A, C	99
SBC A, D	9A
SBC A, E	9B
SBC A, H	9C
SBC A, L	9D
SBC A, n	DEn
SBC HL, BC	ED42
SBC HL, DE	ED52
SBC HL, HL	ED62
SBC HL, SP	ED72
SCF	37
SET 0, (HL)	CBC6
SET 0, (IX + d)	DDCBdC6
SET 0, (IY + d)	FDCBdC6
SET 0, A	CBC7
SET 0, B	CBC0
SET 0, C	CBC1
SET 0, D	CBC2
SET 0, E	CBC3
SET 0, H	CBC4
SET 0, L	CBC5
SET 1, (HL)	CBCE
SET 1, (IX + d)	DDCBdCE
SET 1, (IY + d)	FDCBdCE
SET 1, A	CBCF
SET 1, B	CBC8
SET 1, C	CBC9
SET 1, D	CBCA
SET 1, E	CBCB
SET 1, H	CBCC
SET 1, L	CBCD
SET 2, (HL)	CBD6
SET 2, (IX + d)	DDCBdD6
SET 2, (IY + d)	FDCBdD6
SET 2, A	CBD7
SET 2, B	CBD0
SET 2, C	CBD1
SET 2, D	CBD2
SET 2, E	CBD3
SET 2, H	CBD4
SET 2, L	CBD5
SET 3, (HL)	CBDE
SET 3, (IX + d)	DDCBdDE
SET 3, (IY + d)	FDCBdDE
SET 3, A	CBDF
SET 3, B	CBD8
SET 3, C	CBD9
SET 3, D	CBDA
SET 3, E	CBDB
SET 3, H	CBDC
SET 3, L	CBDD
SET 4, (HL)	CBE6
SET 4,(IX + d)	DDCBdE6
SET 4, (IY + d)	FDCBdE6
SET 4, A	CBE7
SET 4, B	CBE0
SET 4, C	CBE1
SET 4, D	CBE2
SET 4, E	CBE3
SET 4, H	CBE4
SET 4, L	CBE5
SET 5, (HL)	CBEE
SET 5, (IX + d)	DDCBdEE
SET 5, (IY + d)	FDCBdEE
SET 5, A	CBEF
SET 5, B	CBE8
SET 5, C	CBE9
SET 5, D	CBEA
SET 5, E	CBEB
SET 5, H	CBEC
SET 5, L	CBED
SET 6, (HL)	CBF6
SET 6, (IX + d)	DDCBdF6
SET 6, (IY + d)	FDCBdF6
SET 6, A	CBF7
SET 6, B	CBF0
SET 6, C	CBF1
SET 6, D	CBF2
SET 6, E	CBF3
SET 6, H	CBF4
SET 6, L	CBF5
SET 7, (HL)	CBFE
SET 7, (IX + d)	DDCBdFE
SET 7, (IY + d)	FDCBdFE
SET 7, A	CBFF
SET 7, B	CBF8
SET 7, C	CBF9
SET 7, D	CBFA
SET 7, E	CBFB
SET 7, H	CBFC
SET 7, L	CBFD
SLA (HL)	CB26
SLA (IX + d)	DDCBd26
SLA (IY + d)	FDCBd26
SLA A	CB27
SLA B	CB20
SLA C	CB21
SLA D	CB22
SLA E	CB23
SLA H	CB24
SLA L	CB25
SRA (HL)	CB2E
SRA (IX + d)	DDCBd2E
SRA (IY + d)	FDCBd2E
SRA A	CB2F
SRA B	CB28
SRA C	CB29
SRA D	CB2A
SRA E	CB2B
SRA H	CB2C
SRA L	CB2D
SRL (HL)	CB3E
SRL (IX + d)	DDCBd3E
SRL (IY + d)	FDCBd3E
SRL A	CB3F
SRL B	CB38
SRL C	CB39
SRL D	CB3A
SRL E	CB3B
SRL H	CB3C
SRL L	CB3D
SUB (HL)	96
SUB (IX + d)	DD96d
SUB (IY + d)	FD96d
SUB A	97
SUB B	90
SUB C	91
SUB D	92
SUB E	93
SUB H	94
SUB L	95
SUB n	D6n
XOR (HL)	AE
XOR (IX + d)	DDAEd
XOR (IY + d)	FDAEd
XOR A	AF
XOR B	A8
XOR C	A9
XOR D	AA
XOR E	AB
XOR H	AC
XOR L	AD
XOR n	EEn

7 HELPA

This is a versatile utility program written in BASIC for easy modification, to help you edit, load, and run Machine Code. The acronym stands for Hex Editor, Loader, and Partial Assembler. It is based on a ZX81 program in *Machine Code and Better Basic* but has been altered to take advantage of the Spectrum's improved facilities.

1. Type it in, and SAVE it using

   SAVE "helpa" LINE 5

once you're happy there aren't any errors.

2. RUN. It will ask for the amount of memory space to be reserved. First "Do you want nonstandard memory?" with an input. Anything except ENTER will be interpreted as asking for a start value for the Machine Code different from the standard 32000 used throughout this book. It then CLEARs memory, resets RAMTOP, and prints out both the new RAMTOP value and the start address for the Machine Code, as a check.

3. It then asks for the number of data bytes that precede the program area. These must be POKEd later (you might wish to add a routine to do this).

4. It then prints out this value, and the actual start address to be used for the Machine Code.

5. Then it asks for the HEX CODE, giving you a red cursor, a flashing [>] sign.

6. You may now type in (and, using control commands described below, manipulate) Machine Code in hex. As each code group is keyed in, the program strips it of any blank spaces (which means you can use these as you want), splits it into two-character codes, and prints out these in a 10-column display. A final ** delimiter is added on the end of each group; the cursor moves to the end of the code just input.

   For example, the input "□a1□e23□□4" is printed out as

   a1 e2 34 ** [>]

(and so would "a1e234" or "a1□e2□34" be).

7. The next group of codes may now be input, and will be automatically appended immediately after the cursor position. The **s are for the user's convenience only and are ignored when the code is loaded above RAMTOP, so the groups do not *have* to correspond to Z80 instruction groups; however this helps for checking.

8. In addition, "control" instructions may be input. These must be the *first* symbol in their group; subsequent characters are usually ignored except in a few cases described below.

The control commands are:

g	(Go)	Run the Machine Code routine
l	(Load)	Load the Machine Code routine above RAMTOP
m	(Move)	Move cursor forward
n	(Negative)	Move cursor backward
p	(Print)	Print out current hex listing
r	(Relative)	Used for automatic calculation of relative jumps
s	(Save)	Save program
x	(eXcise)	Delete from the listing

More detailed descriptions follow in a more convenient order.

m, n — The command ma, where a is a number, moves the cursor forward a spaces. na moves it back a spaces. It is protected against going off the listing. If a is omitted it will be set to 1.

x — A command xa where a is a number ≥ 0 will delete the next a pairs of characters (including **s) after the cursor. The display will not be affected until p is pressed. If a is omitted it will be set to 1.

To *append* text, just set the cursor to a position immediately in front of where it should go, and enter it. Everything beyond will be shifted up to make room, but only the new text displayed until p is pressed.

p — Prints the current text, and moves the cursor to the top.

s — Saves the program, including the current hex listing. There is an option allowing you to name the program as you wish. To run a saved program, SAVE it *before* typing "*l*" and "g". Then LOAD into BASIC, type GO TO 200 (*not* RUN!!), then "*l*", then "g".

l — Loads the machine code, stripped of superfluous ** delimiters, above RAMTOP, starting at the RAMTOP value you've set; tacks on the final RET command.

g — This runs the routine. Control returns to HELPA (barring a crash!).

r — A useful feature to make *relative* jumps easy: these are a nuisance by hand because of the need to count displacements and put them into 2's complement hex; but nice for programs because of portability. It works like this.

(*a*) Input the machine code routine setting all relative jump sizes to 00.

(*b*) To change to the correct value for a given jump, set the cursor immediately in front of the 00, and delete by hitting "x". Now input rn where the number n is the (decimal) position of the *destination* byte for the jump, read off from the screen display as follows. Number the rows and columns of the hex code array starting at 0, like this:

```
  0 1 2 3 4 5 6 7 8 9
0
1
2
3
.
.
.
```

and set n = xy for row x, column y. (That is, for the byte in row 17, column 5, type r175.) The fact that there are 10 columns makes it easy to read off the numbers. [You could modify the program so that the cursor is moved to the destination, and the jump found from that: in practice this is slower, because of all the cursor-shifting you have to do.]

(*c*) Now press "p" for a revised display with the correct jump size included.

(*d*) Locate the cursor before the next relative jump size, and repeat.

(*e*) Note that the program automatically adjusts for any **s around and gives the 2's complement code needed: if the jump is outside permissible range, it will say so.

(*f*) That may sound complicated, so here's an example, using the Column Scroll routine from Chapter 14. Input in turn the groups of hex codes: the result will be:

Ramtop:	31999
M/C area:	32000
Data:	32000 to 31999
Run USR	32000

HEX CODE:

Ø1 2Ø ØØ ** dd 21 1f 58 ** 3e
15 ** dd 56 2Ø ** dd 72 ØØ **
dd Ø9 ** 3d ** b8 ** 2Ø ØØ **
▷

Where the underlined ØØ is the size of the relative jump to be calculated. (The Data region looks a trifle odd—you could modify the program to produce print-out "Data: none" if the number of data bytes is Ø.)

Using "n" get the cursor in front of the last ØØ like this:

dd Ø9 ** 3d ** b8 ** 2Ø ▷ ØØ **

and then type "x" to delete the ØØ. The instruction here is JRNZ *loop*; and the *loop* starts at the code "dd" on the second line of hex. This is in row 1, column 2 (remember, both start from Ø) so you must input "r12". Do this.

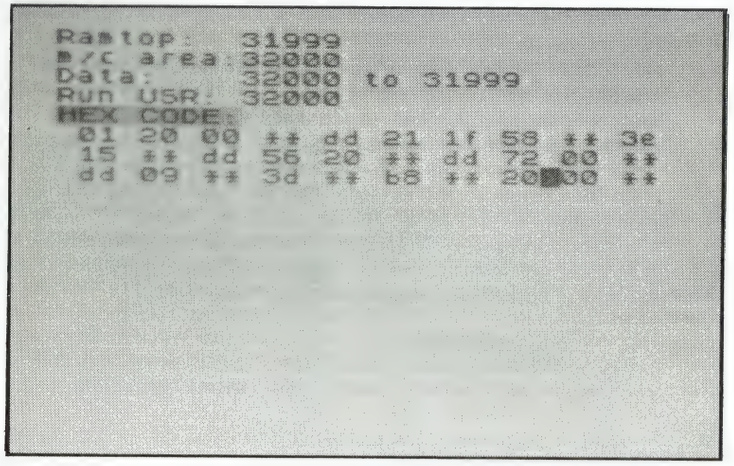


Figure A7.1 HELPA ready to set up a relative jump.

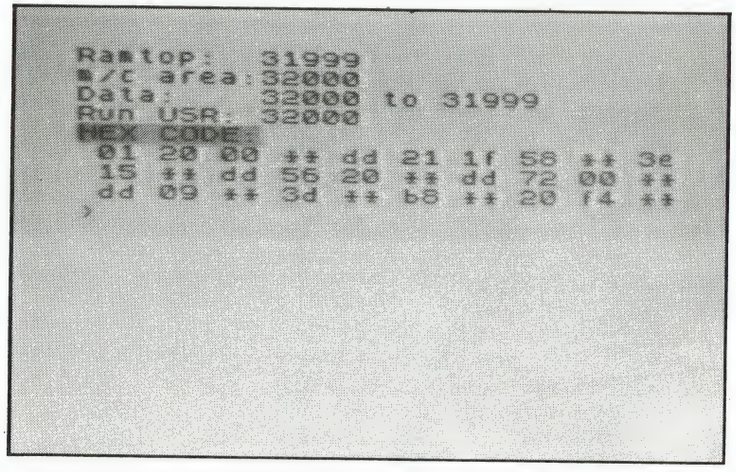


Figure A7.2 The jump address f4 has been written in.

After a moment the screen blanks, then prints out a new listing with the ØØ replaced by f4, the correct relative jump size.

Here's a complete listing.

```
  5 REM helpa © 1982 Ian Stewart & Robin Jones
  7 POKE 23609, 50
 10 INPUT "Do you want nonstandard memory?"; a$
 20 IF a$ <> " " THEN INPUT "START of M/C area?"; rt
 30 IF a$ = " " THEN LET rt = 32000
 40 CLEAR rt - 1
 50 LET rt = PEEK 23730 + 256 * PEEK 23731 + 1
 60 PRINT "Ramtop: □ □"; rt - 1
 70 PRINT "m/c area: □"; rt
 80 INPUT "Number of data bytes?"; d
 90 PRINT "Data: □ □ □ □"; rt; "□ to □"; rt + d - 1
100 PRINT "Run USR: □"; rt + d
110 PRINT PAPER 6; "HEX CODE:"
120 LET ci = 0
130 LET h$ = " "
140 GO SUB 400
150 DIM f(20)
151 LET f(1) = 700
152 LET f(6) = 800
153 LET f(7) = 900
154 LET f(8) = 1000
155 LET f(10) = 1100
156 LET f(11) = 1200
157 LET f(12) = 1300
158 LET f(13) = 1400
159 LET f(14) = 1500
160 LET f(18) = 1600
200 INPUT i$
210 LET a = 0
220 LET a = a + 1
230 IF a > LEN i$ THEN GO TO 300
240 IF i$(a) <> "□" THEN GO TO 220
250 LET i$ = i$(TO a - 1) + i$(a + 1 TO)
260 GO TO 230

300 IF CODE i$(1) > = 103 THEN GO TO 500
310 LET i$ = i$ + "**"
320 LET h$ = h$(TO 2 * ci) + i$ + h$(2 * ci + 1 TO)
330 GO SUB 450
```

```
340  GO SUB 600
350  GO SUB 400
360  GO TO 200

400  REM print cursor
410  PRINT AT 5 + INT(ci/10), 3 * (ci - 10 * INT(ci/10) );
     FLASH 1; INK 2; ">";
420  RETURN

450  REM unprint cursor
460  PRINT AT 5 + INT(ci/10), 3 * (ci - 10 * INT(ci/10) ); "□";
470  RETURN

500  REM keyboard routines
510  GO SUB f(CODE i$(1) - 102) )
520  GO TO 200

600  REM print appended text
610  FOR j = 1 TO LEN i$/2
620  PRINT i$(2 * j - 1 TO 2 * j) + "□";
630  LET ci = ci + 1
640  IF ci = 10 * INT(ci/10) THEN PRINT "□ □";
650  NEXT j
660  RETURN

700  REM go
710  CLS
720  LET y = USR(rt + d)
730  RETURN

800  REM load above ramtop
810  LET h$ = h$ + "c9"
820  LET j = rt + d - 1
830  LET i = -1
840  LET j = j + 1
850  LET i = i + 2
860  IF i > LEN h$ THEN RETURN
870  IF h$(i) = "*" THEN GO TO 850
880  POKE j, 16 * (CODE h$(i) - 48 - 39 * (h$(i) > "9") )
     + CODE h$(i + 1) - 48 - 39 * (h$(i + 1) > "9")
890  GO TO 840
```



```
900  REM move cursor positively
910  GO SUB 450
920  IF LEN i$ = 1 THEN LET cm = 1
930  IF LEN i$ > 1 THEN LET cm = VAL i$(2 TO)
940  LET ci = ci + cm
950  IF ci > LEN h$/2 THEN LET ci = LEN h$/2
960  GO SUB 400
970  RETURN

1000 REM move cursor negatively
1010 GO SUB 450
1020 IF LEN i$ = 1 THEN LET cm = 1
1030 IF LEN i$ > 1 THEN LET cm = VAL i$(2 TO)
1040 LET ci = ci - cm
1050 IF ci < 0 THEN LET ci = 0
1060 GO SUB 400
1070 RETURN

1100 REM print
1110 PRINT AT 5, 0;
1120 FOR r = 1 TO 17: PRINT " [32 spaces] "; : NEXT r
1130 PRINT AT 5, 0; "□";
1140 LET ci = 0
1150 FOR j = 1 TO LEN h$/2
1160 PRINT h$(2 * j - 1 TO 2 * j) + "□";
1170 LET ci = ci + 1
1180 IF ci = 10 * INT(ci/10) THEN PRINT "□ □";
1185 NEXT j
1190 GO SUB 400
1195 RETURN

1300 REM relative jumps
1310 LET jci = VAL i$(2 TO)
1320 LET js = jci - ci - 1
1325 GO SUB 2000
1330 IF js > = -128 AND js < = 127 THEN GO TO 1355
1340 INPUT "Invalid size: ENTER to continue"; a$
1350 RETURN
1355 IF js < 0 THEN LET js = js + 256
1360 LET x1 = INT(js/16)
1365 LET x0 = js - 16 * x1
```

```
1367 LET x$ = CHR$(x1 + 48 + 39 * (x1 > 9)) + CHR$(x0 + 48 + 39 * (x0 > 9))
1370 LET h$ = h$(TO 2 * ci) + x$ + h$(2 * ci + 1 TO)
1375 LET ci = ci + 1
1380 GO SUB 1100
1390 RETURN

1400 REM save
1410 INPUT "SAVE name? Default" "helpa" " "; n$
1420 IF n$ = " " THEN LET n$ = "helpa"
1430 POKE rt + d + LEN h$, 201
1440 SAVE n$ CODE rt, d + LEN h$ + 1
1450 PRINT "To reload use" ' ' "LOAD" " "; n$; " " "□ CODE"
1460 RETURN

1600 REM delete
1610 IF LEN i$ = 1 THEN LET k = 1
1620 IF LEN i$ > 1 THEN LET k = VAL i$(2 TO)
1630 LET h$ = h$(TO 2 * ci) + h$(2 * ci + 2 * k + 1 TO)
1640 RETURN

2000 REM asterisk adjust
2010 IF js < 0 THEN LET w$ = h$(2 * jci + 1 TO 2 * ci)
2020 IF js > = 0 THEN LET w$ = h$(2 * ci + 1 TO 2 * jci)
2030 LET sc = 0
2040 FOR t = 1 TO LEN w$
2050 IF w$(t) = "*" THEN LET sc = sc + 1
2060 NEXT t
2070 IF js < 0 THEN LET js = js + sc/2
2080 IF js > 0 THEN LET js = js − sc/2
2090 RETURN
```


Bibliography

Carr, *Z80 User's Manual*, Beston Publishing Co. Inc.
Logan, *Understanding Your Spectrum*, Melbourne House Group.
Nichols, Nichols, and Rony, *Z80 Microprocessor Programming and Interfacing*, Howard Sams & Co.
Sloan, *Introduction to Minicomputers and Microcomputers*, Addison-Wesley.
Spracklen, *Z80 and 8080 Assembly Language Programming*, Hayden.
Zaks, *Programming the Z80*, Sybex.
Zilog Z80 CPU Programming Reference Card and
Zilog Z80 CPU Technical Manual, Zilog UK Ltd., Nicholson House, Maidenhead, Berks.

Other titles of interest

Easy Programming for the ZX Spectrum £5.95
Ian Stewart & Robin Jones

'. . . will take you a long way into mysteries of the Spectrum; is written with a consistent and humorous hand; and shares the affection the authors feel for the computer'
—*ZX Computing*

Further Programming for the ZX Spectrum £5.95
Ian Stewart & Robin Jones

Computer Puzzles: For Spectrum and ZX81 £2.50
Ian Stewart & Robin Jones

Games to Play on Your ZX Spectrum £1.95
Martin Wren-Hilton

Brainteasers for BASIC Computers £4.95
Gordon Lee

PEEK, POKE, BYTE & RAM!
Basic Programming for the ZX81 £4.95
Ian Stewart & Robin Jones

'Far and away the best book for ZX81 users new to computing'—*Popular Computing Weekly*

'. . . the best introduction to using this trail-blazing micro'—*Computers in Schools*

'One of fifty books already published on the Sinclair micros, it is the best introduction accessible to all computing novices'—*Laboratory Equipment Digest*

Machine Code and better Basic £7.50
Ian Stewart & Robin Jones

The ZX81 Add-On Book £5.50
Martin Wren-Hilton

Easy Programming for the BBC Micro £5.95
Eric Deeson

Further Programming for the BBC Micro £5.95
Alan Thomas

Easy Programming for the Dragon 32 £5.95
Ian Stewart & Robin Jones

Further Programming for the Dragon 32 £5.95
Ian Stewart & Robin Jones

Available from April '83

Spectrum in Education £6.50
Eric Deeson

Programming for REAL Beginners £2.95
Philip Crookall

Shiva Software

Spectrum Special 1 £5.95
Ian Stewart & Robin Jones
A selection of 10 educational games and puzzles.

Spectrum Specials 2 & 3 £5.95 ea
Ian Stewart & Robin Jones

Plus more to come!

Following on from *Easy* and *Further Programming*, we now take you a step beyond: to the realms of Machine Code Programming, specifically for the ZX Spectrum.

Talk to the Z80 microprocessor chip in its native tongue, and save all that time wasted by translating from BASIC.

Explore features of the Spectrum's operating system that can be exploited using Machine Code: Attribute and Display Files, System Variables, and the structure of the BASIC program area.

Sample some of our Machine Code routines:

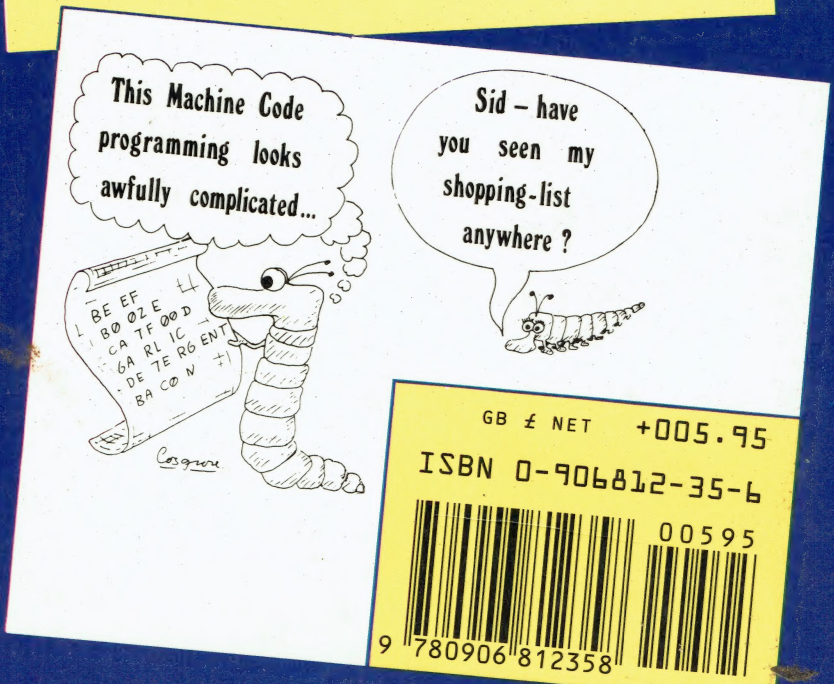
- fancy scrolling
- line renumbering
- rapid searches
- high-speed graphics
- instant colour-changes

Also included are seven appendices to make Machine Code Programming easier; among these: a complete listing of all the Z80 opcodes in alphabetical order, and a BASIC program to edit, load, save, and run Machine Code.

If you own a Spectrum, know nothing about Machine Code, but want to learn – here's your chance!

(Contents apply to both 16K and 48K Spectrum.)

* The earlier book by the same authors, *Machine Code and better Basic*, was serialized in *Popular Computing Weekly*.



GB £ NET +005.95
ISBN 0-906812-35-6
00595
9 780906 812358

Shiva Publishing Limited